
EASS — Class IX course notes (spring 2026), archived

Archived for reference. Superseded by the [Class X course guide](#).

Engineering of Advanced Software Solutions (EASS) · HIT · 27 documents rendered from the Class IX lecture-notes repository (commit 68b1d2f).

Contents

COURSE OVERVIEW

EASS – Engineering of Advanced Software Solutions (2026 Class IX Course Materials)	README.md	3
EASS – Engineering of Advanced Software Solutions (12-Session Plan)	index.md	6
Exercises Overview	exercises.md	10

SESSIONS 01–12

Session 01 – Kickoff and Environment Setup	sessions/session-01.md	13
Session 02 – Introduction to Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST)	sessions/session-02.md	39
Session 03 – FastAPI Fundamentals (Movie Service v0)	sessions/session-03.md	57
Session 04 – Persisting the Movie Service (SQLite + SQLAlchemy)	sessions/session-04.md	81
Session 05 – Multi-DB Modes + Postgres Hardening	sessions/session-05.md	93
Session 06 – Streamlit Dashboards + Typer UX	sessions/session-06.md	101
Session 07 – React/Vite Foundations + Reliability Upgrades	sessions/session-07.md	106
Session 08 – AI Sidecar for Dynamic Compute (Calculator + Codegen + SQL)	sessions/session-08.md	112
Session 09 – Async Recommendation Refresh (EX3 Reliability Checkpoint)	sessions/session-09.md	126
Session 10 – Docker Compose, Redis, and Service Contracts (EX3 Runbook Core)	sessions/session-10.md	134
Session 11 – Security Foundations (EX3 Baseline Controls)	sessions/session-11.md	141
Session 12 – Tool-Friendly APIs & FastMCP Finish Line	sessions/session-12.md	148

OPTIONAL SESSIONS

Optional DuckDB Mini-Lakehouse Lab	sessions/optional/DuckDBMiniLakehouse.md	155
Team Topologies: Organizing Business and Technology Teams for Fast Flow	sessions/optional/TeamTopologies.md	162
Optional MCP Workshop – Weather MCP Server	sessions/optional/mcp.md	163
AI Local SDK README – PydanticAI Gemini & Local Models	sessions/optional/pydantic-ai-local-sdk-readme.md	172

OPTIONAL DEMOS

Streamlit + FastAPI Microservices Demo (Blockchain-Inspired)	optional/blockchain-microservices-demo.md	180
--	---	-----

DEPLOYMENTS

Azure Container Apps Deployment Playbook	deployments/azure-container-apps.md	196
--	-------------------------------------	-----

AI-ASSISTED WORKFLOW

AI-Assisted Coding Workflow (Codex Paradigm)	workflows/ai-assisted/README.md	201
Repo Structure Changes for the AI-Assisted Paradigm	workflows/ai-assisted/repo-structure.md	203
Teaching Guide — AI-Assisted Workflow	workflows/ai-assisted/teaching-guide.md	205
Review Checklist (AI-Assisted Workflow)	workflows/ai-assisted/checklists/review.md	207
Feature Brief Template	workflows/ai-assisted/templates/feature-brief.md	208

TROUBLESHOOTING

EASS Course Troubleshooting Guide	troubleshooting.md	209
-----------------------------------	--------------------	-----

EASS – Engineering of Advanced Software Solutions (2026 Class IX Course Materials)

README.md

This repository contains the fully scripted 12-session plan for the **Engineering of Advanced Software Solutions** course (2026 cohort, Class IX). Each class blends 45 minutes of theory with two 45-minute hands-on blocks, and the entire arc follows a single project: building a movie catalogue with FastAPI, SQLAlchemy, Typer, and Streamlit—no heavy infrastructure or security overhead required.



Quick Start for Instructors

```
git clone https://github.com/EASS-HIT-PART-A-2026-CLASS-IX/lecture-notes.git
cd lecture-notes
```

Primary organization for this cohort: <https://github.com/EASS-HIT-PART-A-2026-CLASS-IX/>.

Open the `docs/` folder (or load the repo in VS Code) to follow any session directly—no static site build is required.

Key documents:

- `docs/index.md` – entry point with links to every session and deadline summary.
- `docs/exercises.md` – specifications and rubrics for the three lightweight exercises.
- `docs/sessions/session-XX.md` – detailed talk tracks, copy/paste code, AI prompt kits, troubleshooting, and verification commands for each class.
- `docs/troubleshooting.md` – quick fixes for common environment issues (uv, imports, Redis, etc.).
- `examples.http` – ready-to-run VS Code REST Client requests for the movie API.
- `docs/workflows/ai-assisted/` – new Codex paradigm for working with AI assistants (briefs, checklists, teaching guide).



Course Highlights

- Sessions 01–04: developer environment (Linux/WSL/macOS), HTTP/REST, FastAPI fundamentals, and the first persistence swap (SQLite via SQLAlchemy) with uv-powered packaging.
- Sessions 05–08: movie service persistence, Streamlit & Typer interfaces, testing/logging, and AI-assisted coding habits.
- Sessions 09–11: optional deep dives (async refreshers, Compose concepts, security vocabulary) for curious students—deliverables stay simple even if you browse these.
- Session 12: polish, demos, and tool-friendly API patterns.
- Cloud onboarding: students use **AWS Educate** at <https://www.awseducate.com/> (self-paced, free, no credit card required) for Compute/Storage/Databases fundamentals.
- Exercises pace with the storyline:
 - **EX1** (assigned Monday, 16/03/2026; due Monday, 30/03/2026 at 23:59 Israel time): ship the FastAPI CRUD service from Session 03 with tests/Docker; adopt Session 04's SQLite upgrade as soon as you're ready so persistence is solved before EX3.
 - **EX2** (assigned Monday, 27/04/2026; due Monday, 18/05/2026 at 23:59 Israel time): deliver a Streamlit dashboard **or** Typer CLI that talks to the EX1 API.

- **EX3** (assigned Monday, 25/05/2026; final submission window opens Monday, 29/06/2026; final due Wednesday, 01/07/2026 at 23:59 Israel time): integrate the API, dedicated persistence layer, and interface into a local multi-service stack (3+ cooperating processes), add one thoughtful improvement, and document the runbook. Everything runs locally; cloud, Docker, and security work are optional extras only.
- **Choose your own domain:** the live sessions use a movie catalogue as the teaching example, but students pick any narrow theme (recipes, books, robotics gear, etc.) and keep it for all three exercises.

Future-facing Engineering Archetypes

Modern software careers are coalescing around four builder profiles, and every lab in this repo intentionally hits each archetype so students can imagine their next role:

1. **Field / Business Engineer (the people person)** – Turns office-hour demos into “this solves your problem” stories and keeps customer value front and center.
2. **DevOps & Infrastructure Engineer (the reliability guru)** – Automates the boring parts: Docker Compose, repeatable env setup, health checks, and log-friendly services.
3. **Full-stack Product Engineer (the end-to-end builder)** – Ships UI, API, and persistence together; FastAPI + Streamlit reps are the backbone of the course.
4. **AI Full-stack Engineer (the intelligence layer)** – Wires agents, retrieval flows, and safe automation; Session 08 plus the optional MCP/DuckDB tracks pave that on-ramp.

Course promise: graduates leave day-one ready for archetypes 3–4, with enough automation muscle memory to be a smart bet for archetype 2 if a team mentors them. By constantly narrating stakeholder impact we keep archetype 1 in the conversation too, so students can explain their builds to humans, not just terminals.

Legacy Materials

Historical slides and Natalie’s notes live under `old-lecture-notes/` :

- `old-lecture-notes/archive/` – previous slide decks and Makefile.
- `old-lecture-notes/notes/` – Natalie’s comprehensive PDF reference.

These are preserved for reference but the new scripted sessions in `docs/` are the canonical teaching materials.

Contributing / Updating

1. Edit the relevant `docs/sessions/session-XX.md` file (each is standalone and self-contained).
2. Run through the verification commands provided in that session (most require `uv run pytest -q` or `curl` checks).
3. Commit changes and push to `main` (the repository is intentionally kept current for instructors).

AI-Assisted Workflow (Codex Paradigm)

The repo is organized around the cycle described in Anindya Chakraborty’s *AI Assisted Coding: Quicker Code Doesn’t Mean Higher Velocity*. Before prompting an assistant, fill out `docs/workflows/ai-assisted/templates/feature-brief.md`, keep diffs under ~150 LOC chunks, and run the review checklist in `docs/workflows/ai-assisted/checklists/review.md`. The complete teaching plan for this workflow lives in `docs/workflows/ai-`

`assisted/teaching-guide.md` ; use it whenever you update a session or add new materials so students learn the same habits.

If you spot an issue or want to suggest an improvement, open a GitHub issue or pull request with the session number in the title (e.g., `Session 05 – clarify rating fixture`).

Have a great semester!

Additional Resources (400+ Question Vault)

- [EASS HIT 100 Questions](#) – Drill site with 400+ prompts to refresh memory and reinforce fundamentals.

EASS – Engineering of Advanced Software Solutions (12-Session Plan)

docs/index.md

Welcome! This site hosts the 12-week plan for the EASS (Engineering of Advanced Software Solutions) course reboot. Everything is organized around short theory bursts followed by two hands-on blocks so undergrads can follow step by step.

- Classes meet on **Mondays** from **02/03/2026** through **01/06/2026** (Semester B 2026).
- Each meeting has **45 min theory + 45 min hands-on build + 45 min practice/extension**.
- Exercises are assigned in class with weekday deadlines that avoid Shabbat and official holiday closures.
- Cloud prerequisite uses **AWS Educate** self-paced training at <https://www.awseducate.com/> (free, no credit card required).

Heads-up from Andrej Karpathy

- Sleep beats all-nighters; aim for ~7½ hours before big work.
- Meet the material early and often—short sessions across days stick best.
- Try problems without notes so you know you can re-create the steps.
- Teach someone else what you learned; explaining makes it click.
- Visit office hours and sessions even if you only have small questions.
- Stop studying alone near the end—compare notes and fill gaps with peers.
- Never hand in early on tests; use every minute to check for silly misses.
- Grades matter, but real projects and references matter more—build things outside class.

Quick Links

- [Course schedule by session](#)
- [Exercise lineup and deadlines](#)
- [AWS Educate \(self-paced, free\)](#)
- [Semester B 2026 calendar \(Jewish calendar aligned\)](#)
- [Optional MCP Workshop](#)
- [Optional DuckDB Mini-Lakehouse Lab](#)
- [Storage engine cheat sheet](#) (SQLite ↔ Postgres ↔ Redis ↔ DuckDB guidance)
- [Legacy slide archives](#)
- [Troubleshooting tips](#)
- [Team Topologies summary](#)
- [DSPy + Pydantic AI agent lab](#)
- [Local Llama.cpp fallback \(Session 08\)](#)
- [Qdrant vector DB stretch \(Session 08\)](#)
- [AI-assisted workflow \(Codex paradigm\)](#)

Semester B 2026 Calendar (Jewish Calendar Aligned)

Event	Date	Notes
First day of Semester B (HIT)	22/02/2026	Institutional opening day
Semester B starts at Faculty of Design	01/03/2026	Faculty opening day
Purim happening (Student Union activity)	01/03/2026	Study break: 13:00-14:00 and 18:00-19:00
Ta'anit Esther	02/03/2026	Study stops from 18:00
Purim	03/03/2026	Holiday (no classes)
Last study day before Passover break	31/03/2026	Zoom learning day
Passover break	01/04/2026- 08/04/2026	Holiday
First study day after Passover break	09/04/2026	Classes resume
Eve of Holocaust and Heroism Remembrance Day	13/04/2026	Classes end at 18:00
Holocaust and Heroism Remembrance Day ceremony	14/04/2026	Study break: 10:00-11:00
Memorial Day ceremony (fallen soldiers and victims of terror)	20/04/2026	Study break: 13:00-14:00
Eve of Memorial Day	20/04/2026	Classes end at 18:00
Memorial Day	21/04/2026	Holiday
Independence Day holiday	22/04/2026	Holiday
Shavuot holiday	21/05/2026- 22/05/2026	Holiday
Last day of Semester B	05/06/2026	Unchanged by institute policy
Make-up days	07/06/2026- 08/06/2026	Makeup for canceled holiday classes; English/general studies exams may occur
Semester B exams	09/06/2026- 03/07/2026	Exam period

Future-facing Engineering Archetypes

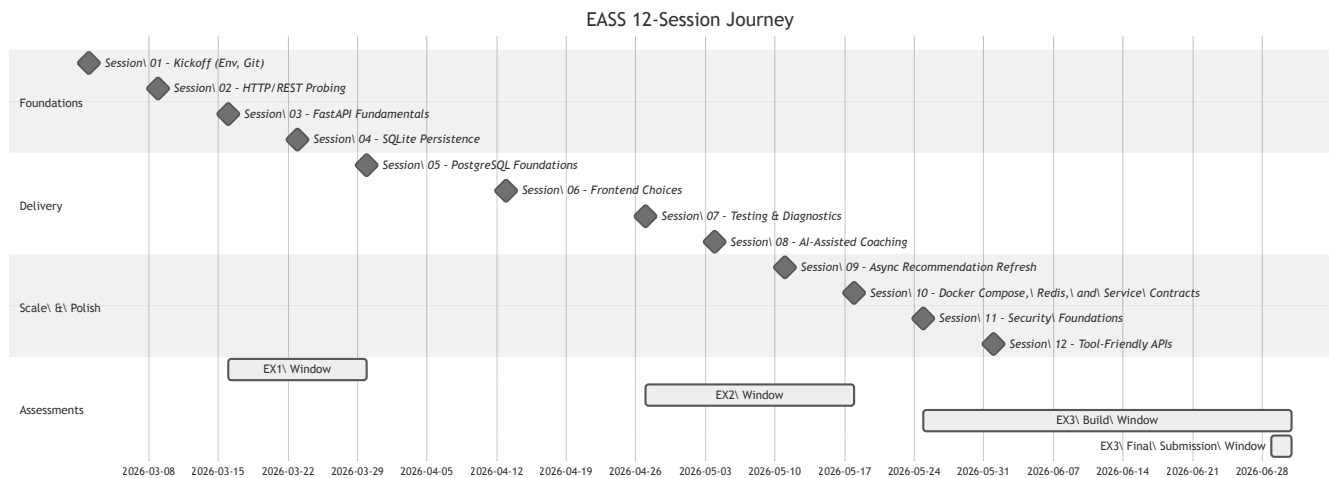
Modern software careers are coalescing around four complementary builder profiles, and EASS intentionally lets students rehearse each mindset while staying within an approachable scope.

1. **Field / Business Engineer (the people person)** – Owns discovery conversations, trims prototypes to the sharpest business value, and turns demos into “this solves your problem” stories.
2. **DevOps & Infrastructure Engineer (the reliability guru)** – Keeps systems observable and shippable; in this course that means Docker Compose fluency, health checks, and deterministic deploys for every lab.
3. **Full-stack Product Engineer (the end-to-end builder)** – Ships UI, API, and data layers together. FastAPI, Streamlit, and shared models appear across sessions so students get repeated reps on this archetype.
4. **AI Full-stack Engineer (the intelligence layer)** – Orchestrates services that learn and act safely. Session 08’s agent labs and the optional MCP/DuckDB tracks show how to wire data contracts for AI-heavy features.

Course promise: graduate students who are day-one ready for archetypes 3–4, while giving them enough automation/infra reps to be a smart bet for archetype 2 if a team decides to mentor them. We keep archetype 1 in view

by narrating every build in stakeholder language, not just in console output.

Visual Roadmap



Course Schedule

1. [Session 01 – Kickoff and Environment Setup](#)
2. [Session 02 – Introduction to HTTP and REST](#)
3. [Session 03 – FastAPI Fundamentals](#)
4. [Session 04 – Persisting the Movie Service \(SQLite + SQLAlchemy\)](#)
5. [Session 05 – PostgreSQL Foundations for the Movie Service](#)
6. [Session 06 – Movie Dashboards with Streamlit & React](#)
7. [Session 07 – Testing, Logging, and Profiling Basics](#)
8. [Session 08 – Working with AI Coding Assistants \(LM Studio, vLLM, or Google AI Studio\)](#)
9. [Session 09 – Async Recommendation Refresh](#)
10. [Session 10 – Docker Compose, Redis, and Service Contracts](#)
11. [Session 11 – Security Foundations](#)
12. [Session 12 – Tool-Friendly APIs and Final Prep](#)

Optional add-ons (extra labs beyond the 12-session arc):

- [MCP Workshop – Weather MCP Server](#) for teams who want to ship MCP-compatible tools after Session 12.
- [DuckDB Mini-Lakehouse Lab](#) for students who want a local analytics sandbox that complements Session 05.

Exercises at a Glance

- **EX1 – FastAPI Foundations (Backend):** assigned **Monday, 16/03/2026** · due **Monday, 30/03/2026 at 23:59 (Israel time)**. Build the FastAPI CRUD API + tests from Session 03; add Session 04's SQLite persistence as soon as you're ready so the service is EX3-ready.
- **EX2 – Friendly Interface (Frontend connected to Backend):** assigned **Monday, 27/04/2026** · due **Monday, 18/05/2026 at 23:59 (Israel time)**. Add a Streamlit dashboard or Typer CLI that calls the EX1 API.
- **EX3 – Full-Stack Microservices Final Project (KISS):** assigned **Monday, 25/05/2026** · final submission window opens **Monday, 29/06/2026** · final due **Wednesday, 01/07/2026 at 23:59 (Israel time)**. Integrate the API,

dedicated persistence layer, interface, and async/Redis worker into a multi-service stack, add one small improvement, and document the runbook + Compose workflow. Everything stays local— `uv run` + `docker compose up` , no cloud hosting required.

- **Choose your own theme:** the sessions demonstrate a movie service, but students pick any narrow domain on Day 1 and carry it through all three exercises.
- Sessions 9–12 continue the main storyline by hardening async flows, multi-service orchestration, security, and tool-friendly polish—each reinforces EX3 readiness while keeping the graded scope local and manageable.

Teaching Philosophy

- Build every example live from the session scripts (no pre-solved `examples/` folders) so changes stay in sync and the cohort can follow the exact steps. Keep snippets tiny and copy/paste friendly when time is tight.
- Repeat concepts using the whiteboard sketches described in each session and the Natalie reference notes in `old-lecture-notes/notes/` .
- Keep optional extras (MCP workshop, DuckDB lab, blockchain demo) clearly labeled so students know the graded work stays lightweight and local.
- Encourage question “warm-ups”: students share what they tried before asking for help.

Happy teaching!

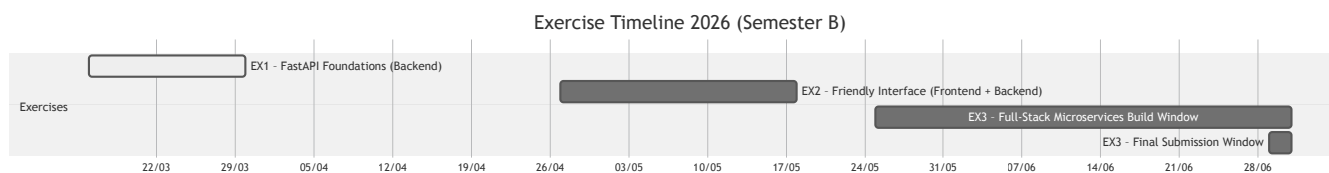
Exercises Overview

docs/exercises.md

Every assignment is intentionally small, runs on a single laptop (Linux, WSL, macOS, or other Unix-like environments), and focuses on getting something working without detours into security hardening or cloud deployment. Students may explore the deeper labs in the session notes, but the required deliverables stay simple and local.

AWS Educate prerequisite: Complete a self-paced **AWS Educate** learning path at <https://www.awseducate.com/> (free, no credit card required) covering **Compute, Storage, and Databases** fundamentals by **Sunday, 29/03/2026 at 23:59 (Israel time)**. Flag blockers early in `#helpdesk`.

Deadline policy: All graded submissions (EX1, EX2, EX3) are scheduled on weekdays and never on Shabbat.



Pick Your Domain Once, Reuse It Everywhere

- Choose a narrow real-world theme (movie catalogue, recipe box, book tracker, indie game backlog, etc.) on Day 1.
- Keep the same theme through EX1–EX3 so each exercise feels like an incremental release of one product.
- All instructions below reference "movies" as the live-coded classroom demo—replace it with your chosen theme while keeping the architectural scope identical.

EX1 – FastAPI Foundations (Backend)

- **Assigned:** Monday, 16/03/2026
- **Due:** Monday, 30/03/2026 at 23:59 (Israel time)
- **Goal:** Ship the core backend microservice for your chosen domain using **FastAPI**, **Pydantic**, and **pytest**. This is the service you'll reuse in every later exercise. Persistence starts in-memory and graduates to SQLite when you integrate Session 04's guidance (mandatory before EX3, optional for the EX1 grading window).
- **Sessions to revisit:** [Session 02 – Introduction to HTTP and REST](#), [Session 03 – FastAPI Fundamentals](#), [Session 04 – Persisting the Movie Service](#).
- **Required Features:**
 - Endpoints: CRUD for your core resource (Session 03 shows `/movies` ; tailor the nouns/fields to your theme). Optional extras belong in stretch goals, not the base grade.
 - Data layer may be in-memory (e.g., simple repository/dict) for the base grade. Plan to adopt Session 04's `SQLModel` + `SQLite` implementation soon after—EX3 expects your backend to talk to a real persistence service with migrations/seeds instead of committing `.db` files.
 - Tests using `pytest` and FastAPI's `TestClient` (cover happy-path listing and create/update/delete).
 - README explaining how to create the uv environment, run the API locally, and execute the tests.

Session guidance: Session 03 is the canonical EX1 implementation (HTTP API, validators, pytest, Docker). Session 04 demonstrates how to layer `SQLModel`/`SQLite` under the same repository interface—treat it as a stretch for EX1 and required prep for EX3's multi-service architecture.

- **Rubric (100 pts):** correctness 40, simplicity/readability 20, tests 20, documentation 20. Bonus (+5) for providing a tiny seed script or `.http` playground tailored to your domain.

EX2 – Friendly Interface (Frontend connected to Backend)

- **Assigned:** Monday, 27/04/2026
- **Due:** Monday, 18/05/2026 at 23:59 (Israel time)
- **Heads-up:** This timeline is positioned after the April memorial/holiday cluster so teams can progress without collision with closures.
- **Goal:** Build a lightweight interface (Streamlit dashboard or Typer CLI) that talks to the EX1 backend for the same domain. Students should be able to exercise the core flows in under a minute from launch.
- **Sessions to revisit:** [Session 05 – PostgreSQL Foundations for the Movie Service](#), [Session 06 – Interface Options](#).
- **Required Features:**
 - Reuse the EX1 API as-is; do not add authentication or security prompts.
 - Allow users to list existing entries and add a new entry in under a minute from launch.
 - Provide one small extra (e.g., mark a favourite item, show a summary metric, or export to JSON/CSV).
 - Document how to run the API and the interface side-by-side on a local machine.
- **Rubric (100 pts):** working flows 40, user guidance (clear copy, prompts, or help text) 25, code clarity 20, documentation 15. Bonus (+5) if tests automate one interface workflow (e.g., calling the Typer command with `CliRunner` or snapshot testing a Streamlit script).

EX3 – Full-Stack Microservices Final Project (KISS)

- **Assigned:** Monday, 25/05/2026
- **Final submission window opens:** Monday, 29/06/2026
- **Final due:** Wednesday, 01/07/2026 at 23:59 (Israel time)
- **Goal:** Finish a tidy local product by joining the EX1 backend, the dedicated persistence layer from Session 04, and the EX2 interface, then adding integrations with at least one additional microservice (LLM tool, reporting worker, notification service, etc.) plus documentation you wish every teammate handed you. Everything must run locally via `uv run / python -m` for scripts and `docker compose up` for the API + Redis + worker stack—no cloud hosting required, but Compose, telemetry, and security guardrails from Sessions 09–12 are now graded expectations.
- **Sessions to revisit:** [Session 07 – Testing and Diagnostics](#), [Session 08 – AI Pairing Tips](#), [Session 09 – Async Recommendation Refresh](#), [Session 10 – Docker Compose, Redis, and Service Contracts](#), [Session 11 – Security Foundations](#), [Session 12 – Tool-Friendly APIs and Final Prep](#). These later sessions deepen reliability, orchestration, security, and release polish—lean on the ones that strengthen your EX3 storyline.
- **Required Pieces:**
 - One Git repository containing at least three cooperating services: the FastAPI backend, a persistence layer (SQLite/SQLModel or equivalent), and a user-facing interface (Streamlit or Typer). Most strong submissions will add a fourth microservice (LLM tool via Pydantic AI, background worker, webhook listener, etc.). Document orchestration in `docs/EX3-notes.md`.
 - A `compose.yaml` plus `docs/runbooks/compose.md` instructions covering how to launch the API, Redis, and async worker, how to verify health/rate-limit headers, and how to run Schemathesis/pytest in CI.
 - Session 09's async refresher deliverable: `scripts/refresh.py` with bounded concurrency, retries, and Redis-backed idempotency plus at least one `pytest.mark.anyio` test. Paste a Logfire or Redis trace excerpt into `docs/EX3-notes.md`.

- Session 11's security baseline: hashed credentials, at least one JWT-protected route with role checks, rotation steps captured in `docs/EX3-notes.md`, and tests that fail when a token is expired or missing scope.
- One enhancement that makes the product nicer without increasing complexity—examples: searchable catalog filtering, weekly recommendation digest generated via a simple SQL query, a CSV import/export command, or a PDF/Markdown summary report.
- Automated tests that cover the enhancement (unit tests, simple integration test, or Typer CLI exercise).
- Local demo script: `uv run python -m app.demo` or `scripts/demo.sh` that walks graders through starting the API, launching the interface, and using the new feature.
- **Rubric (100 pts):** working integration 35, thoughtful enhancement 25, automation/tests 20, documentation/demo assets 20. Bonus (+5) for recording a ≤ 2 min screen capture showing the flow end-to-end (attach a local `.mp4` or shareable link).

 **Scope guardrail:** If a feature smells like production security, massive data ingestion, or multi-service orchestration, it's beyond scope. Keep it local, deterministic, and doable in a weekend.

Submission Guidelines

1. Push code to the designated GitHub Classroom repository (single monorepo for EX1–EX3 is encouraged but not mandatory—clarify structure in your README).
2. Tag releases if requested (e.g., `ex1-final`). Use meaningful commit messages so instructors can review your progression.
3. Include an “AI Assistance” section in each README describing prompts/tools used and how outputs were verified locally.
4. Provide reproducible database setup via migrations or seed scripts; do not commit SQLite `.db` artifacts.
5. Late policy: 48-hour grace period with 10% deduction; communicate early if blockers appear.

Support Channels

- Discord `#helpdesk` channel for quick questions (invite: <https://discord.gg/EYjQrSmF7f>).
- Weekly office hours (posted in class).
- Peer study sessions after select lectures (announced in Discord).

Session 01 – Kickoff and Environment Setup

docs/sessions/session-01.md

- **Date:** Monday, 02/03/2026
- **Theme:** Welcome everyone, set expectations, and make sure every student can create and run a Python project using uv, Git, and Visual Studio Code (VS Code).

Learning Objectives

- Understand course structure, grading, and support channels.
- Install or verify core tools: uv, Git, Python 3.12+, VS Code, Docker Desktop.
- Create a minimal Python project with uv and run tests from the terminal.

Guiding principles: keep everything **KISS** (Keep It Simple, Students) and share questions using the **PARD** format (Problem → Action → Result → Desired) so we can move fast together. All commands assume macOS, Linux, or Windows Subsystem for Linux (WSL); no native Windows PowerShell steps are covered.

You are how you act: forget Rousseau's inner goodness—Franklin reminds us that habits win and you become what you repeatedly do. Even with "vibe coding" and artificial intelligence (AI) helpers at your side, your job is to own the repo, keep the tests green, and make sure the code stays explainable and scalable.

Before Class – Equipment Check (Just-in-Time Teaching, JiTT)

Ask students to complete this 5-minute checklist the night before:

- Run `python3 --version` (or `py --version` on Windows), `uv --version`, `git --version`, and `code --version`. Paste the commands *and* outputs in Discord `#helpdesk` using the **Problem** → **Action** → **Result** → **Desired** template (see below).
- Note any failures or missing commands so we can triage on arrival; do **not** spend hours debugging alone.
- If `uv` is missing, install it ahead of time with `curl -LsSf https://astral.sh/uv/install.sh | sh && exec "$SHELL" -l`.
- This just-in-time teaching checkpoint lets us spend class time on the fun parts instead of long installs.

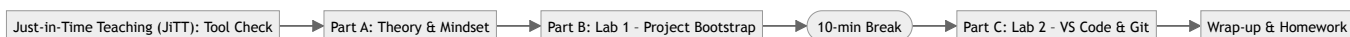
Agenda

Segment	Duration	Format	Focus
PART A – Theory & Motivation	45 min	Talk + Discussion	Course overview, tools landscape, mindset
Warm welcome	5 min	Discussion	Zoom breakouts: one-sentence intros + goals
Course overview	15 min	Talk	Syllabus, grading (3 exercises), single monorepo (recommended) + Exercise 3 (EX3) local final project scope, expectations for collaboration and AI usage
Tool belt briefing	25 min	Talk + quick demos	Why uv, how Git will be used, Visual Studio Code (VS Code) essentials, FastAPI for backends, Docker for deployment, large language models (LLMs) as coding assistants and in projects

Segment	Duration	Format	Focus
Break	10 min	—	Encourage movement – launch 10-minute timer
PART B – Hands-on Lab 1	45 min	Guided coding	Scaffold the starter project, run first test
Environment setup	10 min	Guided	Install verification, create project directory
uv project init	15 min	Live code	Init project, create venv, add pytest
First test	15 min	Live code	Write and run test_math.py
Git foundation	5 min	Live code	Initialize repo, first commit
Break	10 min	—	Encourage movement – launch 10-minute timer
PART C – Hands-on Lab 2	45 min	Guided coding	Explore VS Code, Git basics, README writing
VS Code tour	15 min	Guided	Open project, install extensions, test explorer
Terminal comfort	15 min	Live code	Basic shell commands, file operations
Documentation	10 min	Live code	Create README with run instructions
Git workflow	5 min	Live code	Stage, commit, push (optional GitHub setup)
Wrap-up	5 min	Discussion	Recap, homework reminder

Session Flow at a Glance

1. **Part A – Theory & Motivation:** Kickoff, expectations, tooling overview, and legacy resources.
2. **Part B – Hands-on Lab 1:** Scaffold `hello-uv`, run tests, commit your work.
3. **Break – 10 Minutes:** Launch the shared [10-minute timer](#) and reset.
4. **Part C – Hands-on Lab 2:** VS Code workflow, terminal practice, documentation, and Git basics.
5. **Wrap-up:** Recap key wins and outline homework.



Lab Quick Reference (Printable)

Use these numbered steps when you call out B# (Part B) or C# (Part C). Unless otherwise stated, run tools with `uv run ...` so you never leave the managed environment.

Part B – Bootstrap `hello-uv`

- **B1 – Create project:**

```
mkdir hello-uv && cd hello-uv
```

- **B2 – Project scaffold (no activation needed):**

```
uv init -p 3.12
uv venv --python 3.12
uv add pytest
uv run python --version # should report 3.12.x
```

If `uv` is missing, install it first with `curl -LsSf https://astral.sh/uv/install.sh | sh && exec "$SHELL" -l`.

- **B3 – Write starter tests (editor of your choice):**

```
mkdir -p tests
nano tests/test_math.py
```

(Alternative: `code tests/test_math.py` or `vim tests/test_math.py`.) Paste exactly:

```
def test_addition():
    assert 2 + 2 == 4

def test_subtraction():
    assert 5 - 3 == 2

def test_multiplication():
    assert 3 * 4 == 12
```

- **B4 – Run tests:**

```
uv run pytest -q
```

Expect three dots followed by “3 passed”.

- **B5 – Create `.gitignore` (KISS edition):**

```
cat > .gitignore <<'EOF'
__pycache__/
.pytest_cache/
.venv/
.uv/
.env
EOF
```

- **B6 – Initial commit:**

```
git init
git diff          # working tree check
git add .
git diff --staged
git commit -m "chore: bootstrap hello-uv project"
git log --oneline
```

Optional: add a simple `pytest.ini` to keep output consistent:

```
cat > pytest.ini <<'EOF'
[pytest]
addopts = -q
testpaths = tests
EOF
git add pytest.ini
git commit -m "chore: add pytest config"
```

Part C – Editor, Shell, Documentation

- **C1 – Open VS Code & extensions:**

`code .` (or VS Code → **Open Folder**). Install Python, Pylance, GitLens, REST Client, Docker. Configure pytest discovery if prompted.

- **C2 – Shell reps:**

```
pwd && ls -la && echo "Scratch pad" > notes.txt && git status && rm notes.txt
```

(The same command works in macOS, Linux, and WSL.)

- **C3 – README:**

`code README.md` (or `nano` / `notepad`). Use the provided template.

- **C4 – Docs commit:**

```
git add README.md
git commit -m "docs: add project README"
```

- **C5 (Optional) – Publish via GitHub CLI:**

```
git branch -M main
gh auth status || gh auth login
gh repo create EASS-HIT-PART-A-2025-CLASS-VIII/hello-uv-<handle> \
  --source=, --remote=origin --push --private
```

If `gh` is unavailable, fall back to:

```
git remote add origin git@github.com:EASS-HIT-PART-A-2025-CLASS-VIII/hello-uv-<handle>.git
git push -u origin main
```

Talking Script – Course Overview (First 45 Minutes)

1. **Introduce yourself and the course tone.** "This class is about learning by doing. You will see me live-code and you will copy/paste freely. Questions are always welcome. Trust the process—even if things feel unfamiliar at first, by the end you'll be building real web applications with modern tools."
2. **Set expectations for participation.** "We use Discord for day-to-day help. Post what you tried before asking. Pair up during labs so nobody gets stuck. Join using the Learning Management System (LMS) link (backup: <https://go.eass.dev/discord>) and use the `#helpdesk` channel for all technical questions."
3. **Explain grading.** "We have three large exercises. Each one is assigned on a Monday, and each deadline is set on a weekday that avoids Shabbat and official holiday closures. There are no surprise quizzes. Show up, code along, and you will earn the grade."
4. **Call out the monorepo + KISS pledge.** "You're welcome to keep all three exercises in one Git repo so EX2 naturally builds on EX1 and EX3 pulls everything together—but it's not mandatory. What *is* required: keep every deliverable simple, local, and easy to demo on your laptop. No cloud accounts, no security audits, just clean FastAPI fundamentals that later grow into SQLAlchemy persistence."
5. **Clarify AI policy.** "You may use AI tools like ChatGPT, Claude, Gemini, Cursor, Copilot, LM Studio, or Ollama. You must understand every line you submit, keep a lightweight spec (spec.md or a tessl.io export) in your repo, and when you ask for help share Problem → Action → Result → Desired."
6. **Outline the tool belt.** "Today we confirm Python 3.12+, uv for environments, Git for version control, VS Code for editing, and Docker Desktop for later sessions. Soon we'll add FastAPI for building web APIs and learn to use LLMs both as coding assistants (to help you write code faster) and as components inside your applications (like calling a local LLM endpoint)."

7. **Preview the three KISS exercises.** "You'll ship three small deliverables: EX1 is a FastAPI CRUD API (SQLite layering comes soon after), EX2 is a friendly Streamlit dashboard or Typer CLI, and EX3 stitches them together—with a dedicated persistence service and one thoughtful improvement. Everything runs locally—no cloud or security gauntlets."
 8. **Transition to Part B.** "Let's make sure every laptop can create a project. Follow along exactly; copy/paste saves time."
-

PART A – Theory & Motivation (45 Minutes)

Warm Welcome (5 Minutes)

Instructor Actions:

1. Offer a one-line intro (name + role) and share the session agenda at a glance.
2. Drop learners into small Zoom breakout rooms for two minutes; ask each person to introduce themselves with one sentence (name + current goal).
3. Pull everyone back and reinforce the "trust the process" philosophy: "Even if Git or Docker feel foreign now, by mid-semester you'll be deploying real applications."

Student Actions:

- In your breakout room: state your name and one project you'd like to tackle this semester (web app, API, automation tool, etc.).
- Listen for common interests; we'll use them to form lab buddies later.

Course Overview (15 Minutes)

Instructor Talking Points:

1. Course structure:

- 12 meetings (Mondays), each with theory + two hands-on labs.
- Three major exercises (EX1, EX2, EX3) with weekday deadlines that avoid Shabbat.
- KISS focus: every deliverable runs locally on Linux/WSL/macOS—no cloud hoops or security audits.

2. Grading breakdown:

- EX1 (FastAPI foundations, backend): 30%
- EX2 (Friendly interface): 30%
- EX3 (Full-Stack Microservices Final Project): 40%
- All components must be completed to pass the course.

3. Collaboration & support:

- Discord server: see the LMS announcement for the latest invite (backup short link: <https://go.eass.dev/discord>).
- Use the `#helpdesk` channel for all technical questions and general support.
- When asking for help, follow the **Problem** → **Action** → **Result** → **Desired** pattern: "I tried X, saw result R, and I'm aiming for Y." Screenshots alone are not enough—share the exact command and output.

- Pair programming during labs is encouraged—nobody should be stuck alone.
- Once you join Discord, share your GitHub username and email so I can send you the GitHub Classroom invitations.
- Legacy study aids: clone the archived resources repo (see LMS link) and review `old-lecture-notes/notes/EASS-Complete-Natalie.pdf` (Hebrew) plus `old-lecture-notes/archive/all_slides.pdf` for historical context and solved exercises.

4. AI policy (crucial):

- **You MAY use AI tools** (ChatGPT, Claude, Gemini, Cursor, GitHub Copilot, LM Studio, Ollama, etc.).
- **You MUST understand** every line of code you submit and be ready to explain it during reviews.
- Keep a lightweight specification in your repo (even a `spec.md` or a tessl.io export) so both humans and AI helpers know the goal before you start coding. Update it as requirements evolve.
- Cite AI-assisted changes in your pull requests or README changelog, and never commit secrets—store keys in `.env` (already gitignored).

5. Learning philosophy:

- "This is a hands-on class. You'll see me type commands, and you'll type the same commands."
- "Copy/paste is not cheating—it's efficient. But you must run the code and verify it works."
- "Questions are always welcome. If you're confused, others probably are too."

Student Actions:

- Join Discord now if you have a device handy.
- Note the three exercise deadlines in your calendar.
- Ask any questions about grading or expectations.

Tool Belt Briefing (25 Minutes)

Instructor Talking Points:

The Five Pillars of Our Stack:

1. **Python 3.12+** – Our main programming language.
 - Modern syntax (type hints, `match` statements).
 - Rich ecosystem for web development.
 - When to reach for it: every backend, scripting, and testing task in this course.
2. **uv** – Fast Python environment and dependency manager.
 - Replaces `pip` + `virtualenv` + `pip-tools` with one tool.
 - `uv init` creates `pyproject.toml` (project manifest); `uv add` / `uv sync` write `uv.lock` (exact versions).
 - Can specify Python version: `uv venv --python 3.12` (preferred) or `uv venv --python 3.13` when ready.
 - Tip: Pin the project's minimum Python with `uv init -p 3.12`; matching versions prevent `uv` from recreating your venv unexpectedly.
 - Why not just pip? `uv` is faster (written in Rust), handles lockfiles automatically, manages Python versions, and will be the industry standard soon.
 - Think of it as: cargo (Rust) or npm (Node.js) but for Python.

- Quick demo:

```
uv init -p 3.12          # scaffolds pyproject (requires-python >=3.12)
uv venv --python 3.12    # creates .venv with Python 3.12
uv add pytest            # installs and locks dependency
uv run pytest            # runs command in the environment
uv run python --version  # check Python version in venv
```

- When to reach for it: every time you install or run Python tooling—no more `pip install` in system Python.

3. Git & GitHub – Version control and collaboration.

- Every project starts with `git init`.
- Git tracks changes to your code over time—like "undo" on steroids.
- You'll commit often: "Save early, save often."
- We'll use GitHub Classroom for exercise submissions.
- Basic workflow you'll learn today:

```
git status      # what changed?
git add .       # stage changes
git commit -m "... " # save snapshot with a message
gh repo create ... # create/push via GitHub CLI
```

- Example: if you break something, `git log` shows history and you can go back to any previous version.
- When to reach for it: whenever you change code; push to GitHub for reviews and grading. We standardise on the GitHub CLI (`gh`) for authentication, repo creation, and pull requests.

4. curl – Command-line tool for HTTP requests.

- Imagine a browser without the graphics: you type the request, it prints the response.
- Use it before writing Python so you can see the raw data flowing over HTTP.
- Quick demo:

```
curl https://api.github.com/users/aws      # GET request
curl -s https://api.github.com/users/aws | head -5
```

- `curl` fetches data from the URL.
 - `-s` hides progress noise; piping to `head -5` keeps the output short.
- We'll rely on `curl` to debug APIs all semester.
- When to reach for it: smoke-test an endpoint before you write a client or suspect your code.

5. VS Code – Our code editor.

- Extensions we'll install today: Python, Pylance, GitLens, Docker, REST Client.
- Built-in terminal (no need to switch windows).
- Test explorer (run tests with one click).
- IntelliSense (autocomplete and hints as you type).
- Why VS Code? Free, powerful, works the same on Mac/Windows/Linux, and has extensions for everything.
- When to reach for it: daily editing, debugging, and test runs with visual feedback.

6. FastAPI – Modern web framework (coming in Meeting 3).

- Build REST APIs with automatic validation and documentation.

- Type-safe: catches bugs before they reach production.
- Quick preview:

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

- Why FastAPI? Fast (high performance), modern (uses Python type hints), has built-in OpenAPI docs, and is used by Netflix, Uber, and Microsoft.
- We'll also use **httpx** (not requests) for making HTTP calls—it's modern, supports async, and is the recommended client for FastAPI.
- When to reach for it: building APIs starting Meeting 3, plus anything that needs async/validation out of the box.

7. Docker – Package your app and its environment (Meeting 4).

- "Works on my machine" → "Works everywhere."
- One `Dockerfile` = reproducible deployment.
- Docker Compose = run multiple services together (API + database + reverse proxy).
- Think of it as: a lightweight virtual machine that packages your app with all its dependencies.
- Why Docker? Consistent environments, easy deployment, industry standard, and makes collaboration easier.
- Preview command: `docker run hello-world` (we'll verify this works today).
- When to reach for it: packaging services for Exercise 3 and any time you need parity across machines.

8. LLMs (Large Language Models) – AI as tool and component.

- **As coding assistant:** Use ChatGPT/Cursor to generate boilerplate, write tests, debug errors.
 - Example prompt: "Write a FastAPI endpoint that returns all items from a list."
 - You review, edit, and verify the code.
- **Inside your projects:** Call LLMs via API (OpenAI) or locally (LM Studio).
 - Example: build a chatbot API that uses a local LLM for responses.
 - We'll cover this in Meetings 8 and 12.
- **Key principle:** AI accelerates your work, but you stay in the driver's seat. You must understand every line of code.
- When to reach for it: generating scaffolds, refactoring boilerplate, and building AI-powered features (Meetings 8+).

9. pytest – Testing framework (we'll use today).

- Automatically discovers and runs test functions (functions starting with `test_`).
- Simple assertions: `assert 2 + 2 == 4`
- Why test? Catch bugs early, document expected behavior, enable refactoring with confidence.
- Example:

```
def test_addition():
    result = 2 + 2
    assert result == 4 # test passes if True, fails if False
```

Quick Demos (5 Minutes Each):

1. curl demo:

```
curl -s https://api.github.com/users/aws | uv run python -m json.tool | head -20
```

- Confirms your network works, pretty-prints the JSON, and shows you the shape of the response quickly. Run it inside your project folder so `uv run` uses your 3.12+ environment.

2. uv demo:

```
uv init -p 3.12
uv venv --python 3.12
uv add httpx
uv run python --version
uv run python -c "import httpx; print(httpx.get('https://api.github.com/users/aws').json()
['followers'])"
```

- `uv init` scaffolds a `pyproject.toml` so dependency commands work (set min Python to 3.12).
- `uv venv` creates an isolated Python.
- `uv add` installs a package and locks the version.
- `uv run` executes commands inside that environment (activation is optional; keep using `uv run ...` for consistency).

3. Git demo:

```
git config --global user.name "Your Name"
git config --global user.email "you@school.edu"
git init
echo "# Test Project" > README.md
git add README.md
git commit -m "Initial commit"
git log --oneline
```

- Configure your identity once, then initialize a repo, stage, and commit.
- `git log --oneline` proves the commit is stored.

4. VS Code demo:

- Open Command Palette (Cmd+Shift+P / Ctrl+Shift+P).
- Install Python extension.
- Show integrated terminal (Ctrl+`).
- Show test explorer (beaker icon on left sidebar).

What-If Troubleshooting Chat (5 Minutes):

- Ask the room: "What if `uv --version` fails? What if `python3 --version` shows 3.9? What if `git status` is empty?" Collect answers and reinforce that the Troubleshooting Guide at the end lists the fixes.
- Goal: Normalize encountering errors and remind students to capture Problem → Action → Result → Desired when they ask for help.

Mini Quiz (3 Minutes):

- Rapid-fire questions: "Which command guarantees your tool runs inside the uv environment?", "What does `git add .` do?", "Why keep a spec file in the repo?" Let students answer verbally or via a quick poll to reinforce

retention.

Student Actions:

- Verify you have a terminal open and can run basic commands.
- If you haven't already, install uv (we'll do this together in Part B if needed).
- Open VS Code and locate the terminal and extensions panels.

Transition to Part B: "Now we're going to build a real project together. Open your terminal and follow along. If you get an error, raise your hand—we'll fix it together."

BREAK (10 Minutes)

Encourage movement, grab water, and launch the shared [10-minute timer](#) before diving into Part B. If the link fails, run `python -c "import time; time.sleep(600)"` or set a phone timer for 10 minutes.

PART B – Hands-on Lab 1: Build Your First Python Project (45 Minutes)

Goal: By the end of this lab, every student will have a working Python project with a passing test, managed by uv and tracked with Git.

Environment Setup (10 Minutes)

Instructor Actions:

1. Invite students to run **Quick Reference B1** (create project directory) and confirm they are inside it.
2. Walk through the uv installation check from **B2** step-by-step: verify `uv --version`, demonstrate the installer command if anyone is missing it, then show how restarting the shell surfaces the new binary.
3. Remind everyone to validate Git identity (`git config --global ...`) before moving on, modelling the Problem → Action → Result → Desired pattern if someone's identity is unset.

Student Checklist:

- ☐ Terminal is open
- ☐ uv is installed (run `uv --version` to verify)
- ☐ Git is configured with your name and email

Buddy Check (5 Minutes):

- Pair up with the person next to you and confirm each checkbox together.
- If one of you is stuck, post in `#helpdesk` using the Problem → Action → Result → Desired format so an instructor or classmate can jump in.

uv Project Initialization (15 Minutes)

Instructor Live Coding (students follow along exactly):

1. Execute **Quick Reference B2** together. Narrate what each command does: `uv init` scaffolds metadata, `uv venv` isolates Python, `uv add` pins dependencies.
2. If anyone activates the environment out of habit, point out the `(.venv)` prompt but emphasise that the class standard is to rely on `uv run ...` instead.
3. Close with a group `uv run pytest --version` to prove the dependency landed.

Student Checkpoint:

- Raise your hand if you see "pytest 8.x.x" or similar output.
- If you see an error, stop and ask for help.

First Test (15 Minutes)

Instructor Live Coding:

1. Guide students through **Quick Reference B3-B5**. Narrate the vim keystrokes (insert, paste, `Esc :wq`) and explain pytest output symbols (`.` vs `F`).
2. After the intentional failure demo, debrief why red tracebacks are learning signals, not disasters. Encourage them to share PARD-formatted questions when they get stuck.

Student Task Card:

- Run `uv run pytest -q` and verify you see `3 passed`.
- If you see failures, check your code matches exactly.
- Raise your hand if tests don't pass.

Micro Demos (Optional, 90 Seconds Each)

- **Red → Green:**

```
printf 'def test_fail(): assert 2 + 2 == 5\n' > tests/test_fail.py
uv run pytest -q
sed -i.bak 's/5/4/' tests/test_fail.py # macOS BSD sed; Linux users can drop .bak
uv run pytest -q
```

Use this to normalise fast iteration: fail, inspect, fix, rerun.

- **See the wire:**

```
curl -s https://api.github.com/users/aws \
| uv run python -m json.tool \
| head -20
```

Emphasise that we “look at the wire” before writing clients.

Git Foundation (5 Minutes)

Instructor Live Coding:

1. Execute **Quick Reference B6** and pause after each sub-command to emphasise the why: `.gitignore` keeps noise out of history, `git add .` stages, the commit message documents intent.
2. Show `git log --oneline` and explain how it becomes the time machine for the project.

Student Checkpoint:

- Everyone should see one commit in `git log --oneline`.
- If you see an error about user.name or user.email, run the config commands from earlier.

Part B Wrap-Up (2 Minutes)

Instructor Summary: "You just created a Python project from scratch using modern tools. You have:

- A virtual environment (`.venv`)
- A project manifest (`pyproject.toml`)
- A lockfile (`uv.lock`)
- Automated tests (`tests/test_math.py`)
- Version control (`.git`)

This is the foundation for every project you'll build in this course."

Student Success Criteria:

- ☐ `uv run pytest -q` shows 3 passed
- ☐ `git log --oneline` shows at least one commit
- ☐ You understand what each command does (if not, ask now!)

Stretch Goal (Optional):

- Create a new branch `feature/division-test`, add a division test that guards against zero division, commit it, and keep it ready for your first pull request next week.

BREAK (10 Minutes)

Stand up, stretch, grab water. When we return, we'll explore VS Code and improve our project.

Launch a shared [10-minute timer](#) (or a local 10-minute timer if the link is blocked) so everyone returns on time.

PART C – Hands-on Lab 2: VS Code, Git Workflow, and Documentation (45 Minutes)

Goal: Get comfortable with VS Code as your development environment, practice Git basics, and document your project professionally.

VS Code Tour (15 Minutes)

Instructor Live Demo (students follow along):

1. Trigger **Quick Reference C1** to open VS Code (either `code .` or File → Open Folder). If the shell command is missing, demonstrate installing it via the Command Palette.
2. **Explorer sidebar (left side):**
 - Show the file tree
 - Expand/collapse folders

- Create new files/folders by right-clicking

3. Install essential extensions:

- Click Extensions icon (or Ctrl+Shift+X / Cmd+Shift+X)
- Install these one by one:
 - a. **Python** (Microsoft) – syntax highlighting, debugging
 - b. **Pylance** (Microsoft) – type checking, IntelliSense
 - c. **GitLens** – Git superpowers (blame, history)
 - d. **REST Client** – test HTTP APIs (we'll use this in Meeting 2)
 - e. **Docker** (Microsoft) – manage containers (we'll use this in Meeting 4)

4. Open integrated terminal:

- Press Ctrl+` (backtick) or View → Terminal
- You can have multiple terminal tabs
- The terminal starts in your project directory automatically

5. Test Explorer:

- Click the beaker icon on the left sidebar (Testing)
- You should see `tests/test_math.py` with three tests
- Click the green play button next to a test to run it
- Click the play button at the top to run all tests

6. Run a test from the explorer:

- Click the play button next to `test_addition`
- See the checkmark when it passes
- Click on the test to see the code

Instructor Tips:

- Walk around the room to verify everyone has VS Code open
- Help students who get stuck on extension installation
- Show how to split the editor (drag a file tab to the side)

Student Checklist:

- ☐ VS Code is open with your `hello-uv` project
- ☐ Python, Pylance, and GitLens extensions are installed
- ☐ Test Explorer shows your three tests
- ☐ You can run a test from the Test Explorer

Terminal Comfort (15 Minutes)

Instructor Live Coding (students type along):

Time check: if you're running late, compress this block to the first command group (`pwd + ls + file create/delete`) and move the rest to homework.

1. Follow **Quick Reference C2** as a guided practice: run `pwd`, `ls -la`, create `notes.txt`, inspect it, then clean it up. Explain what each command reveals about the project.
2. Demonstrate `git log --oneline --graph` and optional aliases once the workspace is back to a clean state.

Additional Terminal Commands to Demonstrate:

```
# Count lines in a file:
wc -l tests/test_math.py

# Search for text in files:
grep "def test" tests/test_math.py

# Show disk usage:
du -sh .venv

# Show Python version:
python3 --version
uv run python --version

# Show environment info:
uv run python -c "import sys; print(sys.executable)"
```

Share a printed or PDF **terminal cheat sheet** (Mac shortcuts on the left, Windows/WSL on the right) so students can reference commands after class.

Student Task:

- Type each command and observe the output.
- Ask questions if anything is confusing.
- Experiment: create a file, read it, delete it.

Documentation (10 Minutes)

Instructor Live Coding:

1. Execute **Quick Reference C3** and model the `vim` workflow again: insert mode, paste template, `Esc :wq`.
2. **Write documentation** (students copy this structure):

Hello uv Project

A minimal Python project demonstrating modern tooling with `uv`, `pytest`, and `Git`.

Setup

1. Clone this repository (or create a new project):

```
```bash
git clone <your-repo-url>
cd hello-uv
```

- ii. Create the uv virtual environment (Python 3.12+; matches `pyproject.toml`'s `requires-python`) and install dependencies:

```
uv venv --python 3.12
uv sync
```

## How to Run Tests

Run all tests (quiet mode):

```
uv run pytest -q
```

## Project Structure

---

```
hello-uv/
├── .venv/ # Virtual environment (not committed)
├── tests/ # Test files
│ └── test_math.py
├── pyproject.toml # Project manifest
├── uv.lock # Locked dependencies
└── README.md # This file
```

## Tools Used

---

- **uv**: Fast Python package manager and environment manager
- **pytest**: Testing framework
- **Git**: Version control

## Next Steps

---

- Add more complex tests
- Create application code in `app/` directory
- Add FastAPI endpoints (Meeting 3)
- Containerize with Docker (Meeting 4)

## Author

---

Your Name - EASS Course, Semester B 2026

3. **Save the file** (Cmd+S / Ctrl+S).

**Instructor Explanation:** "A good README answers three questions:

1. What is this project?
2. How do I set it up?
3. How do I use it?

Every project you submit must have a README. This is your first impression."

**Student Task:**

- Create your own `README.md` with at least:
  - Project title
  - Setup instructions
  - How to run tests
  - Your name

## Git Workflow (5 Minutes)

### Instructor Live Coding:

1. Run **Quick Reference C4** to stage and commit `README.md`. Pause to highlight the difference between working tree, staging area, and commit history.
2. For teams ready to publish, follow **Quick Reference C5**. Show the live GitHub repo so students see the end-to-end flow (create repo under the organisation, add remote, push, verify).
  - Highlight the GitHub CLI flow: `gh auth login`, `gh repo create ... --push`, and later `gh pr create` for reviews.

### Collaboration Preview (2 Minutes):

- Explain that future exercises will use feature branches and pull requests (branch → commit → open PR → code review → merge).
- Show one slide or live GitHub screenshot so students see where comments and reviews will appear.

**Instructor Note:** If students don't have GitHub accounts or SSH keys set up, make this optional homework. The important part is they understand local Git workflow.

### Student Checkpoint:

- ☐ `README.md` exists and has content
- ☐ `git log --oneline` shows two commits
- ☐ (Optional) Project is pushed to GitHub

## Part C Wrap-Up (2 Minutes)

**Instructor Summary:** "You now have a complete development workflow:

1. Write code in VS Code
2. Run tests to verify it works
3. Commit changes with Git
4. Document your work in README

This is the workflow you'll use for all three exercises."

### Student Success Criteria:

- ☐ Comfortable opening projects in VS Code
- ☐ Can run tests from Test Explorer
- ☐ Can use basic terminal commands
- ☐ Have created a professional README
- ☐ Understand `git add`, `git commit`, `git status`

---

## Optional Preview: Typer, Pydantic, FastAPI, SQLAlchemy

If you finish early, install a small bundle and explore what's ahead:

```
uv add typer pydantic fastapi sqlmodel uvicorn
```

- **Typer (CLI skeleton):**

```
cli.py
import typer

app = typer.Typer()

@app.command()
def greet(name: str = "world"):
 print(f"Hello, {name}!")

if __name__ == "__main__":
 app()
```

Run with `uv run python cli.py greet --name Alice`.

- **Pydantic (validated input):**

```
validate.py
from pydantic import BaseModel, ValidationError

class Item(BaseModel):
 name: str
 price: float

try:
 print(Item(name="pen", price="1.5").model_dump())
except ValidationError as exc:
 print(exc)
```

- **FastAPI (10-line POST):**

```
app.py
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
 name: str
 price: float

@app.post("/items")
def create(item: Item):
 return {"ok": True, "total": item.price}
```

Start it with `uv run uvicorn app:app --reload --port 8000` and test via:

```
curl -s -X POST localhost:8000/items \
-H "content-type: application/json" \
-d '{"name":"pen","price":1.5}' \
| uv run python -m json.tool
```

- **SQLModel (typed rows in SQLite):**

```
db_demo.py
from sqlmodel import SQLModel, Field, Session, create_engine, select

class Product(SQLModel, table=True):
 id: int | None = Field(default=None, primary_key=True)
 name: str
```

```

 price: float

engine = create_engine("sqlite:///memory:")
SQLModel.metadata.create_all(engine)

with Session(engine) as session:
 session.add(Product(name="pen", price=1.5))
 session.commit()
 print(session.exec(select(Product)).all())

```

Treat these as teasers—you'll master them in later sessions.

---

## Wrap-Up Script (5 Minutes)

---

### Instructor Final Thoughts:

1. **Celebrate progress:** "Today you built a working Python project from scratch and learned the tools you'll use all semester. That's a huge accomplishment!"

2. **Review what you accomplished:**

-  Created a Python project with `uv`
-  Wrote and ran automated tests with `pytest`
-  Tracked your work with `Git`
-  Documented your project with `README.md`
-  Set up VS Code as your development environment

3. **Homework and next steps:**

- **Tonight:**
  - Join Discord (LMS link or <https://go.eass.dev/discord>)
  - Share your GitHub username and email in `#helpdesk` so I can send you invitations
- **This week:**
  - Keep your `hello-uv` repo handy and re-run `uv run pytest -q` to stay sharp.
  - Jot down one improvement you want to add to the movie API once we build it next week.
  - (Optional) Push your `hello-uv` project to GitHub once you get the GitHub Classroom link.
- **Before next class:**
  - Make sure Docker Desktop is installed and running
  - Run `docker run --rm hello-world` to verify Docker works
  - Review HTTP basics if you're rusty (we'll cover it in depth next week)

4. **Preview of Meeting 2:** "Next Monday we dive into HTTP and REST APIs. You'll learn how browsers and servers talk, and you'll build your first HTTP client using `htpx`. Bring your laptops charged and your `hello-uv` project ready."

5. **Q&A:** "Any questions about today's material, the exercises, or expectations?"

6. **Final reminder:** "Remember: trust the process. If something doesn't make sense today, it will make sense by Meeting 4. Keep coding, keep asking questions in `#helpdesk`, and help each other out on Discord."

7. **Resource wall:** "Snapshot these links—Discord, GitHub Classroom FAQ, Troubleshooting Guide, tessl.io spec template, and the terminal cheat sheet PDF—so you can reference them after class."

**Student Exit Ticket (Optional):** Post in Discord `#helpdesk` :

- One thing you learned today
- One thing you're confused about
- One thing you're excited to build

---

## Materials Checklist (For Instructor)

---

### Before Class:

- ☐ Python 3.12+ installed on all lab machines
- ☐ uv installed (Astral installer or package manager; verify with `uv --version`)
- ☐ Git installed and verified
- ☐ VS Code installed with Python extension
- ☐ Docker Desktop installed (for verification only today)
- ☐ Discord invite link ready (see LMS or backup <https://go.eass.dev/discord>)
- ☐ Slides or talking points prepared for Part A
- ☐ Example `hello-uv` project on your machine for demo
- ☐ Lab Quick Reference + terminal cheat sheet shared (print or PDF)

### During Class:

- ☐ Share Discord link in chat/on board
- ☐ Remind students to share GitHub username and email in `#helpdesk`
- ☐ Walk around during hands-on labs to help students
- ☐ Take note of common errors for troubleshooting guide

### After Class:

- ☐ Post session recording (if recorded) to Canvas
- ☐ Share troubleshooting notes in Discord
- ☐ Send GitHub Classroom invitation links to students who shared their GitHub usernames
- ☐ Prepare starter code for Meeting 2 (HTTP client examples with `httpx`)

---

## Support Blocks (Run as Needed)

---

### Windows WSL Setup Guide (Self-Contained)

*Share with Windows participants who are still on PowerShell-only environments.*

1. In elevated PowerShell:

```
wsl --install
wsl --set-default-version 2
```

2. Inside the new Ubuntu shell:

```
sudo apt update && sudo apt -y upgrade
sudo apt -y install build-essential curl git python3 python3-venv
```

3. Install the VS Code "Remote – WSL" extension, open the project inside WSL, and verify `/mnt/c` visibility so students know where Windows files live.

## Docker Sanity Check (5 Minutes)

Assign as homework once uv/Git are stable.

```
docker run --rm hello-world
```

Expect the success message; on Windows remind them to enable WSL Integration in Docker Desktop first.

---

## PARD Help Template (Copy/Paste into #helpdesk)

Pronounced "Pard". Always include the exact output so we can help quickly.

```
Problem: <short description>
Action(s): <exact commands you ran>
Result: ``<full output / error>``
Desired: <what you expected or are trying to achieve>
```

Providing all four lines saves everyone time and gives you sharper answers.

---

## Troubleshooting Guide

### Issue: `uv: command not found`

#### Solution:

```
Mac/Linux:
curl -LsSf https://astral.sh/uv/install.sh | sh
(optional) Homebrew users: brew install uv

After installing, restart your terminal or run:
exec "$SHELL" -l
```

If you still prefer `pip install uv`, do it inside a disposable virtual environment so the binary lands on your PATH predictably.

## Issue: `git commit` fails with "Please tell me who you are"

### Solution:

```
git config --global user.name "Your Name"
git config --global user.email "you@school.edu"
```

## Issue: `uv venv` fails with permission or missing module errors

### Solution (macOS/Linux/WSL):

```
python3 --version # ensure 3.12.x
python3 -m ensurepip --upgrade
python3 -m pip install --upgrade pip
sudo apt install -y python3.12-venv # Ubuntu/WSL only
uv venv --python 3.12
```

Most failures come from missing `python3.12-venv` or an older Python binary on the PATH.

## Issue: Python version is too old (< 3.12)

### Solution:

- **macOS:** `brew install python@3.12` then ensure `python3 --version` reports 3.12 ( `brew info python@3.12` for linking tips).
- **Ubuntu/Debian (including WSL):** `sudo apt update && sudo apt install -y python3.12 python3.12-venv` . If multiple versions exist, run `sudo update-alternatives --config python3` on lab machines.
- **Other distros:** install Python 3.12 via your package manager or pyenv; align all machines to 3.12 for this course.

## Issue: Tests don't run in VS Code Test Explorer

### Solution:

1. Make sure Python extension is installed
2. Open Command Palette (Cmd+Shift+P / Ctrl+Shift+P)
3. Type "Python: Configure Tests"
4. Select "pytest"
5. Select "tests" as the test directory

## Issue: Docker Desktop + WSL integration pitfalls

### Solution:

1. Verify WSL 2 is installed: `wsl --version`
2. In Docker Desktop → Settings → General, ensure "Use WSL 2 based engine" is checked
3. Under Resources → WSL Integration, enable your Linux distro
4. Restart Docker Desktop, then rerun `docker run --rm hello-world`

## Issue: `ModuleNotFoundError: No module named 'httpx'`

### Solution:

```
If you haven't initialized a project yet:
uv init -p 3.12
uv add httpx
uv run python -c "import httpx; client = httpx.Client();
print(client.get('https://api.github.com/users/zozo123').status_code)"
```

- Make sure you run the command inside the project directory so `uv run` picks up the virtual environment.
- If you already installed dependencies, run `uv sync` before retrying your `python -c` script.
- For multi-line scripts, pipe a here-doc into Python instead of packing everything into one line:

```
uv run python - <<'PY'
import httpx

client = httpx.Client()
n = p = 0
while True:
 response = client.get(
 "https://api.github.com/users/zozo123/repos",
 params={"per_page": 100, "page": p + 1},
).json()
 if not response:
 break
 n += len(response)
 p += 1
print(n)
PY
```

## Issue: `uv add ...` times out

### Solution:

```
uv --network-timeout 60 add pytest
```

Raising the timeout usually resolves slow mirror responses. Persistent failures typically indicate a proxy/firewall configuration problem—capture it with PARD and escalate.

## Issue: Can't push to GitHub (permission denied)

### Solution for SSH:

```
Generate SSH key:
ssh-keygen -t ed25519 -C "you@school.edu"

Start ssh-agent:
eval "$(ssh-agent -s)"

Add key:
ssh-add ~/.ssh/id_ed25519

Copy public key:
cat ~/.ssh/id_ed25519.pub
Paste this into GitHub → Settings → SSH and GPG keys → New SSH key
```

### Solution for HTTPS:

- Use a Personal Access Token instead of password
- GitHub → Settings → Developer settings → Personal access tokens → Generate new token
- Use the token as your password when pushing

### Issue: VS Code `code` command doesn't work

#### Solution:

- **macOS:** Open VS Code → Command Palette (Cmd+Shift+P) → "Shell Command: Install 'code' command in PATH"
- **Linux/WSL:** Add VS Code's bin directory to your PATH, e.g. `export PATH="$PATH:/usr/share/code/bin"`

---

## Extension: Advanced Git Setup (Optional, for Fast Students)

---

If students finish early, guide them through:

### 1. Create a `.gitignore` file:

```
Create .gitignore in project root:
cat > .gitignore << 'EOF'
Python
__pycache__/
*.py[cod]
*$py.class
*.so
*.Python

Virtual environments
.venv/
venv/
ENV/

IDE
.vscode/
.idea/
*.swp
*.swo

Testing
.pytest_cache/
.coverage
htmlcov/

OS
.DS_Store
Thumbs.db

uv
.uv/
EOF

git add .gitignore
git commit -m "chore: add .gitignore for Python project"
```

## 2. Create a feature branch:

```
Create and switch to a new branch:
git checkout -b feature/add-division-test

Add a new test:
(edit tests/test_math.py to add test_division)

git add tests/test_math.py
git commit -m "test: add division test"

Switch back to main:
git checkout main

Merge the feature:
git merge feature/add-division-test
```

## 3. View commit history with more detail:

```
git log --oneline --graph --decorate --all
git log --stat
git show HEAD
```

---

## Student Success Criteria (Final Checklist)

By the end of Session 01, every student should be able to:

- ☐ **Environment:** Create a Python virtual environment with `uv venv`
- ☐ **Dependencies:** Add packages with `uv add <package>`
- ☐ **Testing:** Write and run tests with `pytest`
- ☐ **Version Control:** Initialize a Git repo, stage files, and commit changes
- ☐ **IDE:** Open a project in VS Code and use the Test Explorer
- ☐ **Terminal:** Navigate directories, create/delete files, run commands
- ☐ **Documentation:** Create a README with setup and usage instructions
- ☐ **Troubleshooting:** Know where to ask for help (Discord `#helpdesk`)
- ☐ **Artifacts:** Project folder includes `.venv`, `pyproject.toml`, `README.md`, and `tests/test_math.py`

If a student cannot do any of the above, schedule office hours before Meeting 2.

---

## AI Prompt Kit (For Instructors and Students)

Use these prompts with ChatGPT/Cursor to generate additional content:

### For Instructors:

1. **Generate troubleshooting scenarios:**

"Create five common error scenarios students might encounter when setting up a Python project with uv and pytest on Windows, with detailed solutions."

## 2. Create quiz questions:

"Write ten multiple-choice questions testing understanding of Git basics (init, add, commit, status) appropriate for first-time users."

## 3. Generate coding exercises:

"Create three progressive coding exercises for students learning pytest: 1) basic assertions, 2) parameterized tests, 3) fixtures."

## 4. Verify installations:

"Write a numbered checklist for students to confirm uv, pytest, git, VS Code, and Docker Desktop are installed correctly on Windows with WSL."

## 5. Welcome message template:

"Draft a short Discord welcome message reminding students to post what they tried before asking for help."

## For Students:

### 1. Understand an error message:

"I got this error when running `uv add pytest`: [paste error]. What does it mean and how do I fix it?"

### 2. Learn a command:

"Explain what `git commit -m 'message'` does, and why the `-m` flag is important. Give me three example commit messages."

### 3. Debug code:

"My pytest test is failing with this error: [paste error]. Here's my code: [paste code]. What's wrong?"

### 4. Generate boilerplate:

"Write a pytest test function that checks if a string is a palindrome. Include docstring and edge cases."

---

## Quick Reference (External Resources)

---

For quick lookups during class:

### Google searches:

- `uv python package manager tutorial`
- `pytest getting started guide`
- `git basics cheat sheet`
- `VS Code Python tutorial`

- `install WSL 2 Windows 11 step by step`

### Official documentation:

- uv: <https://github.com/astral-sh/uv>
- pytest: <https://docs.pytest.org/>
- Git: <https://git-scm.com/doc>
- VS Code Python: <https://code.visualstudio.com/docs/python/python-tutorial>

### Video resources:

- "Git explained in 100 seconds" ( Fireship on YouTube)
- "Python virtual environments explained" (Corey Schafer)
- "VS Code setup for Python" (Microsoft)

### ChatGPT prompt:

- "Create a five-point checklist to confirm uv, pytest, git, and Docker Desktop are installed correctly on a student laptop."

---

### End of Session 01 Guide

# Session 02 – Introduction to Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST)

docs/sessions/session-02.md

- **Date:** Monday, 09/03/2026
- **Theme:** Demystify HTTP request/response flows and capture reusable probes that will shape Exercise 1 (EX1).

## Core Concepts (Own These by End of Session)

### 1. Client–Server Over HTTP

- **Client** initiates (browser, curl, httpx, your Typer CLI). **Server** listens and responds (FastAPI `/movies`).
- **Daily life:** You order food (client), kitchen cooks only when asked (server).
- **Test yourself:** Point at code and say "this is client" / "this is server" without hesitation.

### 2. Request–Response Cycle

Every HTTP exchange is:

```
Request = method + URL (path + query) + headers + optional body
Response = status code + headers + optional body
```

**Anchor example:**

```
GET /movies?limit=10 HTTP/1.1
X-Trace-Id: 123abc

200 OK
X-Trace-Id: 123abc
[{"id": 1, "title": "Alien"}]
```

**Goal:** Read raw HTTP and label every part without help.

### 3. Stateless Server

- Each request carries everything the server needs (auth token, params).
- Server forgets you between requests unless you give it a token.
- **Why it matters:** Easier scaling, simpler debugging.

### 4. REST: Resource-First (Not RPC)

- **Nouns** in paths (`/movies`, `/movies/42`), **verbs** in HTTP methods.
- **Good:** `GET /movies`, `POST /movies`, `DELETE /movies/42`
- **Smelly:** `POST /createMovie`, `GET /getAllMovies`
- **Test:** Hide methods and look only at URLs. Do they look like *things* (✅) or *actions* (❌)?

## 5. CRUD ↔ HTTP Verbs + Idempotency

Action	Verb	Idempotent?	Example
Create	POST /movies	No (retry creates duplicate)	Body: {"title": "Alien"}
Read	GET /movies, GET /movies/42	Yes (safe, cacheable)	No body
Update (replace)	PUT /movies/42	Yes (retry safe)	Body: full object
Update (partial)	PATCH /movies/42	Sometimes	Body: {"year": 2011}
Delete	DELETE /movies/42	Yes (repeat = same end state)	Returns 204 even if already gone

**Idempotent** = doing it twice gives same result. Think of a simple math function:

- $f(x) = |x|$  (absolute value) is idempotent because once the number is positive, applying  $f$  again keeps it the same:  $f(f(-3)) = f(3) = 3$ .
- $f(x) = x + 1$  is **not** idempotent because  $f(f(x)) = x + 2$ .

In HTTP terms, calling `PUT /movies/42` twice leaves the resource in the same state (like  $f(x) = 1$ ), but calling `POST /movies` twice creates two records (like  $f(x) = x + 1$ ).

**Safe** = doesn't change data (e.g., GET); **idempotent** = same end state even if repeated (e.g., PUT, DELETE).

## 6. Gateway (Reverse Proxy)

- Sits **in front of** app servers (nginx before FastAPI).
- Routes traffic, adds headers, terminates SSL, rate-limits, logs.
- **Does NOT own business logic** (that's the server's job).
- **Daily life**: Reception desk logs visitors and directs them, but doesn't do the actual work.
- **Your stack**: `curl/TypeR → nginx → FastAPI`

## 7. Trace ID (Observability First-Class Citizen)

- **Rule**: Every request carries `X-Trace-Id`. Gateway and server propagate it. Logs and errors include it.
- **Daily life**: Package tracking number used at every hop.
- **EX1 contract**:

```
{
 "status": 404,
 "error": "resource_not_found",
 "detail": "Movie 42 not found",
 "trace_id": "123abc"
}
```

## 8. Status Code Vocabulary (Must Own)

- 200 OK, 201 Created, 204 No Content
- 400 Bad Request, 404 Not Found, 422 Validation Failed, 429 Too Many Requests
- 500+ Server error (never leak stack traces)

Link every EX1 error path to one of these.

## 9. Tools = Same Concept, Different Ergonomics

- **curl**: raw, scriptable, shows HTTP anatomy
- **VS Code REST Client**: visual, good for `.http` recipes
- **httpx + Typer**: programmable, testable, becomes EX1 test harness

Same mental model: client crafts request, reads response.

---

## Learning Objectives

- Explain client-server architecture and the gateway pattern using nginx and Docker MCP as concrete examples.
- Break down every component of an HTTP request and response (method, path, query, headers, cookies, body, status code).
- Map CRUD (Create, Read, Update, Delete) operations to HTTP verbs using the `/movies` resource you'll ship in EX1.
- Contrast "resource-first" REST patterns with ad-hoc remote procedure call (RPC) flows so students can spot good API design.
- Call public APIs with `httpx`, `curl`, and the VS Code REST Client, piping outputs through `python -m json.tool` for fast inspection.
- Automate exploratory requests behind a Typer CLI so EX1 debugging is repeatable.
- Document baseline contracts ( `.http` files + error format checklist) that evolve into EX1 tests.

---

## Concept Jumpstart (Explain Like I'm New)

### Core architecture (read these in order):

- **Client**: Anything that starts the conversation (browser tab, `curl`, `httpx`, Typer CLI). It crafts the HTTP request and decides what to do with the response.
- **Server**: The program waiting for requests (FastAPI in EX1). It listens on a port, runs business logic, and sends responses.
- **HTTP**: The simple language both sides speak—verbs (GET/POST/PUT/DELETE), paths ( `/movies` ), headers ( `Accept`, `X-Trace-Id` ), and optional body JSON.
- **Gateway (reverse proxy)**: Sits between clients and servers to handle cross-cutting concerns (logging, SSL, caching, rate limiting) without changing app code. Nginx is our gateway.

### REST fundamentals:

- **Resources**: REST says "treat everything like a noun." `/movies/42` is a resource; the HTTP verb describes the action.
- **CRUD mapping**: Create → POST, Read → GET, Update → PUT/PATCH, Delete → DELETE. We'll drill this into muscle memory before EX1.
- **Idempotency**: Safe to retry. `GET` and `PUT` are idempotent; `POST` usually isn't (retry creates duplicate resources).

### Observability:

- **Trace ID**: A short ID the client generates so every hop (client → gateway → server → database) can log the same value. It's our "package tracking number."

## Tooling landscape:

- **curl** : One-off command-line requests, great for spot-checks and shell scripts.
- **VS Code REST Client**: GUI helper for visually constructing requests and sharing `.http` snippets.
- **httpx** + **Typer**: Python client we can wrap in a CLI, add tests to, and reuse across EX1 debugging.

## Proxy taxonomy:

- **Forward proxy**: Hides clients (e.g., corporate proxy that anonymizes your laptop when browsing).
- **Reverse proxy (gateway)**: Hides servers (e.g., nginx in front of FastAPI that routes traffic and adds headers).
- **Mental model**: Forward = "I represent you to the world." Reverse = "I represent your app to the world."

## Before Class – REST Warm-Up (JiTT)

Create a fresh Session 02 workspace at `~/eass-25/playground` so nobody depends on leftovers from Session 01:

```
mkdir -p ~/eass-25/playground
cd ~/eass-25/playground
uv init -p 3.12 # scaffold pyproject + uv.lock tied to Python 3.12
uv python install 3.12 # download the interpreter if it is missing
uv python pin 3.12 # ensure every uv run uses Python 3.12
uv run python --version # double-check the version reads 3.12.x
uv add httpx pydantic typer pytest
```

**Reminder:** Keep every install command in this session as `uv add <package> --skip manual version specifiers` and let `uv.lock` capture the exact pins.

Post the command output in Discord `#helpdesk` using **Problem** → **Action** → **Result** → **Desired** if anything fails.

Install the VS Code **REST Client** extension and verify connectivity:

```
curl https://httpbin.org/get?ping=preflight | python -m json.tool
```

Share the pretty-printed JSON screenshot in your lab thread so everyone confirms the tooling.

**AWS Educate:** Complete a self-paced learning path at <https://www.awseducate.com/> (free, no credit card required) covering **Compute, Storage, and Databases** fundamentals by **Sunday, 29/03/2026 at 23:59 (Israel time)**; flag blockers early.

**Sanity check:** Re-run `uv run pytest -q` inside `~/eass-25/playground` so the testing muscle stays warm.

## Admin & Homework Checkpoints

- **Roster sync:** Post your GitHub username **and** best contact email in the Discord `#roster-sync` channel before Session 02 ends so we can add you to the GitHub org, Classroom repos, and mailing list without delays.
- **AWS Educate (self-paced, free, no credit card):** complete the Compute/Storage/Databases fundamentals track at <https://www.awseducate.com/> by **Sunday, 29/03/2026 at 23:59 (Israel time)**; log blockers in `#helpdesk` immediately.
- **EX1 – FastAPI Foundations (Backend):** due **Monday, 30/03/2026**. Revisit the `rubric` while you build the Typer probes in this session.

- **EX2 – Friendly Interface (Frontend connected to Backend):** opens **Monday, 27/04/2026**, due **Monday, 18/05/2026**. Skim the [brief](#) now so today's HTTP probes feed directly into your UI/CLI plan.
- **EX3 – Full-Stack Microservices Final Project (KISS):** kicks off **Monday, 25/05/2026**; final submission window opens **Monday, 29/06/2026** and closes **Wednesday, 01/07/2026**. Keep a running log in your repo for the [final project checklist](#).

## Agenda

Segment	Duration	Format	Focus
Part A – Recap & AWS Educate checkpoint	5 min	Guided discussion	Round-robin: automation/aliases from Session 01? AWS Educate Compute/Storage/Databases progress?
Part A – Client-server & gateway concepts	10 min	Talk + board	Architecture fundamentals: client, server, gateway roles.
Part A – nginx+Docker gateway demo	10 min	Live demo	Run nginx as reverse proxy, observe request/response flow.
Part A – HTTP mechanics, status codes, and error design	15 min	Talk + whiteboard	Methods, status codes, headers, trace IDs, resource naming, idempotency.
Part A – Docker MCP gateway example	5 min	Live demo	Show MCP tooling pattern that mirrors HTTP client design.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided coding</b>	<b>Build reusable HTTP client with <code>httpx</code> + Typer CLI.</b>
Break	10 min	—	Launch <a href="#">10-minute timer</a> and stretch.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided practice</b>	<b>Capture <code>.http</code> recipes, validate error contracts, prep Schemathesis.</b>
EX1 briefing & backlog	10 min	Talk + Q&A	Scope reminder, backlog items (pagination, rate limiting, Schemathesis).

## Part A – Theory & Live Demos (45 Minutes)

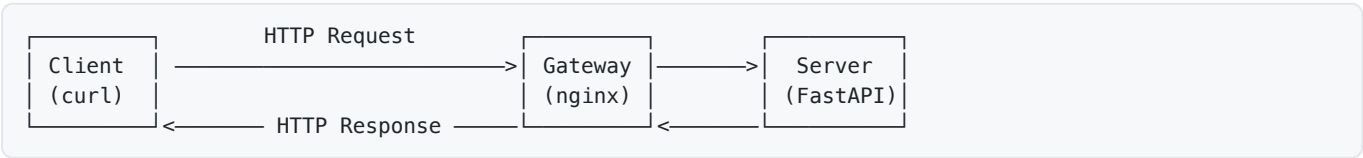
### 1. Recap & AWS Educate checkpoint (5 min)

Round-robin: automation/aliases from Session 01? AWS Educate Compute/Storage/Databases progress?

### 2. Client-server architecture fundamentals (10 min)

Kick off with an everyday analogy: the diner (client) reads the menu and places an order, the waiter (gateway) relays orders and checks IDs, and the kitchen (server) prepares food. Same idea with HTTP—clients send requests, gateways clean them up, servers cook the response.

Draw on board:



### Client responsibilities:

- Initiates requests (browser, curl, httpx, mobile app)
- Constructs HTTP request (method, path, headers, body)
- Handles responses (parse JSON, retry on errors, cache)

### Server responsibilities:

- Listens on port (80/443 for HTTP/HTTPS)
- Routes requests to handlers (FastAPI endpoints in EX1)
- Generates responses (status code, headers, JSON body)

### Gateway/Reverse Proxy responsibilities:

- Sits between clients and servers
- Load balancing (distribute traffic across multiple servers)
- SSL termination (decrypt HTTPS, forward HTTP internally)
- Request routing (path-based: `/api/*` → FastAPI, `/static/*` → file server)
- Caching (store frequently accessed responses)
- Rate limiting (throttle abusive clients)
- Header injection (add `X-Forwarded-For`, `X-Trace-Id`)

**Key insight:** In EX1, your Typer CLI is the **client** and FastAPI is the **server**. You can swap persistence under that contract later and bring in an nginx gateway for health checks and logging as the stack grows.

## 3. nginx+Docker gateway demo (10 min)

**Goal:** Show HTTP request/response flow through a real gateway before students build their own client.

Create temporary demo directory:

```
mkdir -p ~/demo-nginx && cd ~/demo-nginx
```

### Create nginx config that proxies httpbin.org:

```
filepath: ~/demo-nginx/nginx.conf
events {}

http {
 server {
 listen 8080;

 location / {
 # Gateway behavior: forward requests to upstream server
 proxy_pass https://httpbin.org;

 # Inject trace ID for observability
 proxy_set_header X-Gateway-Trace $request_id;

 # Preserve original client info
 proxy_set_header X-Forwarded-For $remote_addr;
 proxy_set_header Host httpbin.org;
 }
 }
}
```

```
}
}
```

### Run nginx in Docker:

Make sure Docker Desktop (or dockerd) is running before you execute the container commands. If Docker is stopped, start it now so the following steps succeed.

```
docker run --rm -d \
 --name demo-gateway \
 -p 8080:8080 \
 -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf:ro \
 nginx:alpine

Wait for startup
sleep 2

Test gateway: your request → nginx:8080 → httpbin.org
curl http://localhost:8080/get?gateway=demo | python -m json.tool

Check nginx logs to see request flow
docker logs demo-gateway

Cleanup
docker stop demo-gateway
```

### Ask class while reviewing output:

- "Where does the request originate?" (curl client)
- "What does nginx do?" (forwards to httpbin.org, adds headers)
- "Why would we use a gateway in EX1?" (centralize logging, SSL, rate limiting)

**Connection to EX1:** FastAPI stays the direct endpoint for EX1. When you build the Compose stack in Session 10, you can drop nginx in front for health checks and structured logging without changing the routes.

### Forward vs reverse proxy cheat sheet:

- **Forward proxy:** Lives near the client. Example: corporate proxy that hides every developer's laptop IP when browsing the public internet.
- **Reverse proxy (gateway):** Lives near the server. Example: nginx in front of FastAPI that hides the app containers and decides which one should receive the request.
- **Mental model:** If the proxy represents you (the client) to the world, it's forward. If it represents your app (the server) to the world, it's reverse.

### Use-case checklist for gateways/reverse proxies:

- Add headers ( `X-Trace-Id` , `X-Forwarded-For` ) without changing app code.
- Retry or circuit-break misbehaving backends.
- Terminate SSL once, then send plain HTTP on your private network.
- Serve `/static/*` locally while forwarding `/api/*` to FastAPI.

## 4. HTTP mechanics, status codes, and error design (15 min)

### Draw full stack with trace propagation:

```

Browser → nginx (gateway) → FastAPI → SQLite → Redis
 ↓ ↓ ↓ ↓ ↓
trace_id trace_id trace_id trace_id trace_id

```

### Status codes:

- 2xx: success (200 OK, 201 Created, 204 No Content)
- 3xx: redirects (301 Moved Permanently, 304 Not Modified)
- 4xx: client errors (400 Bad Request, 401 Unauthorized, 404 Not Found, 422 Validation, 429 Too Many Requests)
- 5xx: server errors (500 Internal Server Error, 503 Service Unavailable)

### Critical headers:

- `Accept` , `Content-Type` : content negotiation (JSON vs XML)
- `Authorization` : auth tokens
- `X-Trace-Id` : correlation across services (inject now, visualize in logs)
- `X-Forwarded-For` : original client IP (gateway adds this)
- `Retry-After` : rate limiting (429 responses)

### Standard error envelope for EX1 (introduce now, implement in your FastAPI server):

Every error response must include these four fields:

```

{
 "status": 404,
 "error": "resource_not_found",
 "detail": "Movie with ID 42 does not exist",
 "trace_id": "demo-1234"
}

```

### Why this structure?

- `status` : Mirrors HTTP code for consistency
- `error` : Machine-readable identifier for programmatic handling
- `detail` : Human-friendly explanation for debugging
- `trace_id` : Links error to logs across all services

**Anti-pattern:** Leaking stack traces or framework exceptions in production responses. All 5xx errors must sanitize internal details.

### HTTP Methods:

- GET: safe, idempotent, cacheable (fetch resources)
- POST: creates resources (not idempotent—retry creates duplicates)
- PUT: idempotent update (safe to retry)
- PATCH: partial update
- DELETE: removes resource

### CRUD ↔ HTTP verbs (anchor everything to `/movies`):

CRUD action	HTTP verb	Example request	Expected response	Notes
Create	POST /movies	Body: {"title": "Inception", "year": 2010}	201 Created + Location: /movies/42	Server generates ID.
Read (list)	GET /movies?limit=10	No body	200 OK + array	Safe to repeat; caches well.
Read (single)	GET /movies/42	No body	200 OK + object, or 404	Must return error envelope on 404.
Update/Replace	PUT /movies/42	Body: full movie object	200 OK or 204 No Content	Idempotent—retry safe.
Update/Patch	PATCH /movies/42	Body: {"year": 2011}	200 OK	Only changed fields. Not always idempotent.
Delete	DELETE /movies/42	No body	204 No Content	Should return 204 even if called twice (idempotent).

### REST resource naming:

- ✓ /movies (collection), /movies/{id} (item), /movies/{id}/ratings (sub-resource)
- ✗ /getMovie , /createMovie (RPC smell—verbs belong in HTTP method, not path)
- ✓ PUT /movies/{id} is idempotent (same request twice = same result)
- ✗ POST /movies/{id} breaks REST conventions (POST to item is create, not update)

### REST vs RPC with /movies example:

#### REST approach:

```
GET /movies/42 # Read
POST /movies # Create
PUT /movies/42 # Update
DELETE /movies/42 # Delete
```

#### RPC approach (avoid):

```
POST /getMovie # Everything is POST
POST /createMovie
POST /updateMovie
POST /deleteMovie
```

#### REST wins because:

- Clients learn pattern once, predict new endpoints ( /ratings , /users )
- HTTP caching works (browsers cache GET, not POST)
- Gateway knows which verbs to rate-limit (POST/PUT/DELETE are write ops)

**Discussion point:** Every request in EX1 must generate or accept a trace ID so we can correlate logs, errors, and cache hits across all layers.

## 5. Docker MCP gateway pattern (5 min)

This is a stretch demo—skip if time is tight or Docker MCP tooling isn't installed. The core client/server/gateway lessons are already covered by nginx.

**Goal:** Show how gateway pattern applies to AI tooling, reinforcing the client-server-gateway mental model.

Docker MCP (Model Context Protocol) exposes AI tools through a gateway that multiple clients can call:

```
Pull DuckDuckGo search tool (MCP server)
docker pull mcp/duckduckgo

Enable it in gateway
docker mcp server enable duckduckgo

Run gateway (like nginx, but for AI tools)
docker mcp gateway run --servers=duckduckgo &

Client calls gateway to list available tools
docker mcp tools ls

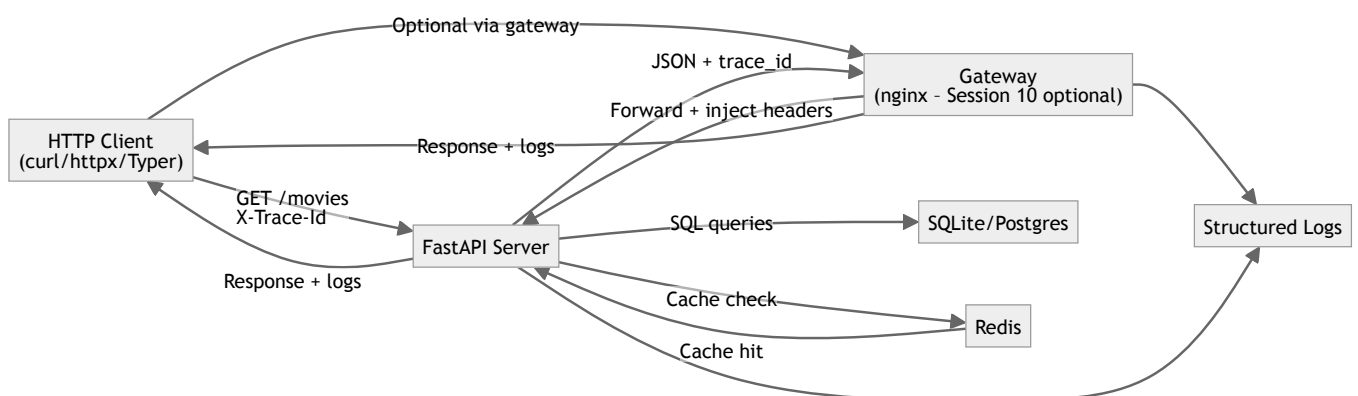
Client calls search tool through gateway
docker mcp tools call search query="Yossi Eliaz" max_results=3
```

### Analogy breakdown:

- **Client:** `docker mcp tools call` (like curl/httpx in Lab 1)
- **Gateway:** `docker mcp gateway` (like nginx, routes requests to correct tool)
- **Server:** `duckduckgo` MCP server (like FastAPI, implements business logic)

**Connection to EX1:** Your Typer CLI in Part B is the client and FastAPI is the server. You can layer SQLite/SQLModel beneath it and add an nginx gateway for logging/health without changing the routes. Same architecture pattern at different scales.

## 6. Preview EX1 architecture (5 min)



**Contract:** Every component must accept and propagate `X-Trace-Id` for end-to-end observability.

**Next build step:** Build the FastAPI **server** that your Typer client calls. Implement the `/movies` CRUD endpoints from the table above and ensure every error matches the envelope structure. You can tuck SQLite/SQLModel under the same routes later; an nginx gateway can wait until you need centralized logging/health.

## Part B – Hands-on Lab 1 (45 Minutes)

**Goal:** Build an HTTP client that demonstrates client-side responsibilities from the architecture diagram.

### Flow for the lab:

1. **Observe:** Re-run `curl https://httpbin.org/get` and highlight method (GET), path (/get), headers (Accept), response status (200).
2. **Translate:** Replace curl with `httpx` in Python so we can add Pydantic validation and pytest.
3. **Automate:** Wrap `httpx` in Typer so `uv run python -m app.cli echo` becomes your reusable CRUD probe.
4. **Test:** Write one pytest that verifies the client handles the server's JSON contract correctly.

### Scaffold inside the Session 02 workspace

Use the dedicated `~/eass-25/playground` workspace you just created (rerun the JiTT commands now if you skipped them).

```
cd ~/eass-25/playground
uv sync
mkdir -p app tests
touch app/__init__.py app/http_client.py app/cli.py
```

If dependencies are missing:

```
uv add httpx pydantic typer
```

### Implement `app/http_client.py`

Create the HTTP client with trace ID generation:

```
filepath: app/http_client.py
from __future__ import annotations

import logging
import uuid
from typing import Any

import httpx
from pydantic import BaseModel

logger = logging.getLogger(__name__)

class PingResponse(BaseModel):
 """Validate httpbin.org echo response.

 This models the server's response format—Pydantic ensures
 the server contract matches our expectations.
 """
 args: dict[str, Any]
 headers: dict[str, Any]
 url: str
 origin: str

def ping(city: str = "Haifa") -> PingResponse:
```

```

"""HTTP client that calls httpbin.org with trace header.

Client responsibilities demonstrated:
- Generate trace ID for observability
- Construct HTTP request (method, URL, headers, params)
- Handle response (validate JSON, check status, log)
- Propagate errors with context

Args:
 city: Query parameter to echo back

Returns:
 Validated response model

Raises:
 httpx.HTTPStatusError: On 4xx/5xx responses
 httpx.TimeoutException: On network timeout
"""
trace_id = uuid.uuid4().hex[:8]

httpx.Client is the HTTP client component
with httpx.Client(timeout=httpx.Timeout(10.0)) as client:
 response = client.get(
 "https://httpbin.org/get", # Server endpoint
 params={"city": city},
 headers={
 "Accept": "application/json",
 "X-Trace-Id": trace_id, # Client-generated trace ID
 },
)
 response.raise_for_status() # Fail fast on errors

Validate server response matches expected contract
payload = PingResponse.model_validate(response.json())

logger.info(
 "httpbin echo",
 extra={"trace_id": trace_id, "status_code": response.status_code},
)
return payload

```

### Discussion checkpoint:

- "What happens if httpbin.org returns 500?" (raise\_for\_status throws exception)
- "Why short trace ID?" (easier to grep logs during debugging)
- "How does Pydantic help?" (catches contract violations before they reach application logic)

### Wire Typer CLI (app/cli.py)

```

filepath: app/cli.py
import json
import logging
from typing import Optional

import typer

from .http_client import ping

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")

app = typer.Typer(help="HTTP client probes for EX1 preflight")

```

```

@app.command()
def echo(city: str = typer.Argument("Haifa"), pretty: bool = True) -> None:
 """Call httpbin.org and print validated JSON.

 Examples:
 uv run python -m app.cli echo
 uv run python -m app.cli echo Tel-Aviv --no-pretty
 """
 response = ping(city=city)
 if pretty:
 typer.echo(json.dumps(response.model_dump(), indent=2))
 else:
 typer.echo(response.model_dump_json())

@app.command()
def headers(city: str = "Haifa") -> None:
 """Print headers echoed back by httpbin.org.

 Check for X-Trace-Id in output to confirm trace propagation.
 """
 response = ping(city=city)
 for key, value in response.headers.items():
 typer.echo(f"{key}: {value}")

def main(argv: Optional[list[str]] = None) -> None:
 app(standalone_mode=True)

if __name__ == "__main__":
 main()

```

## Run and verify client behavior

```

uv run python -m app.cli echo --city Tel-Aviv
uv run python -m app.cli headers --city London

```

### Ask class to identify:

- Where is the client? (Typer CLI wrapping httpx)
- Where is the server? (httpbin.org)
- Where would a gateway fit? (between client and httpbin, like nginx demo)

**Check:** Highlight `X-Trace-Id` in headers output. Ask: "How would you propagate this to FastAPI logs when you build the server?"



**Quick win:** Structured JSON output with trace IDs confirms your client can drive HTTP servers repeatably.

**Map to CRUD:** Today's `echo` command is the "Read" action (GET). Next, add commands for:

- `create` → `POST /movies` with JSON body
- `list` → `GET /movies` with optional filters
- `update` → `PUT /movies/{id}` with full object
- `delete` → `DELETE /movies/{id}` and verify 204 response

Same `httpx` client, just different verbs and paths.

## Add pytest smoke test

```
filepath: tests/test_ping.py
from app.http_client import ping

def test_ping_default_city():
 """Verify httpbin.org connectivity and response contract.

 This is a client-side test: validates our client correctly
 handles the server's response format.
 """
 response = ping()
 assert response.args["city"] == "Haifa"
 assert response.url.startswith("https://httpbin.org")
 # You can mock this later to avoid flaky network tests
```

```
uv run pytest -q tests/test_ping.py
```

## Part C – Hands-on Lab 2 (45 Minutes)

**Goal:** Capture reusable request patterns and validate the error envelope contract you'll implement in your FastAPI server.

### Create `.http` request collection

```
<!-- filepath: requests.http -->
GET with trace ID (client → server)
GET https://httpbin.org/get?city=Haifa
Accept: application/json
X-Trace-Id: demo-1234

POST form data (client sends payload to server)
POST https://httpbin.org/post
Content-Type: application/x-www-form-urlencoded
X-Trace-Id: demo-5678

name=EASS
cohort=2026

HEAD for metadata only (client asks for headers, no body)
HEAD https://httpbin.org/get
Accept: application/json

Trigger 404 error (server error response)
GET https://httpbin.org/status/404
Accept: application/json
X-Trace-Id: demo-404

###
```

**Walkthrough:** Send each request from VS Code (Cmd/Ctrl+Alt+R). Point out client role (constructing request) vs server role (generating response).

### Validate error envelope contract

**Goal:** Create test cases that your FastAPI server must pass.

```
<!-- filepath: docs/contracts/http-errors.md -->
Standard Error Envelope (Server-Side Contract)
```

All EX1 error responses must follow this structure so clients can parse errors programmatically:

```
```json
{
  "status": <HTTP status code>,
  "error": "<machine_readable_identifier>",
  "detail": "<human explanation>",
  "trace_id": "<echo inbound X-Trace-Id or generate new>"
}
```

Test Cases for the FastAPI server

404 Resource Not Found:

```
GET http://localhost:8000/movies/99999
X-Trace-Id: test-404
```

```
# Expected response:
```

```
# Status: 404
```

```
# Body:
```

```
{
  "status": 404,
  "error": "resource_not_found",
  "detail": "Movie with ID 99999 does not exist",
  "trace_id": "test-404"
}
```

422 Validation Error:

```
POST http://localhost:8000/movies
Content-Type: application/json
X-Trace-Id: test-422
```

```
{
  "year": 2010
}
```

```
# Expected response (missing required field 'title'):
```

```
# Status: 422
```

```
# Body:
```

```
{
  "status": 422,
  "error": "validation_failed",
  "detail": "Field 'title' is required",
  "trace_id": "test-422"
}
```

500 Internal Server Error:

```
GET http://localhost:8000/movies?trigger_error=true
X-Trace-Id: test-500
```

```
# Expected response (sanitized, no stack trace):
```

```
# Status: 500
```

```
# Body:
```

```
{
  "status": 500,
```

```
"error": "internal_error",
"detail": "An unexpected error occurred. Check logs with trace_id.",
"trace_id": "test-500"
}
```

Server Implementation Checklist

- ☐ All 4xx and 5xx responses follow envelope structure
- ☐ Never leak stack traces or framework internals in `detail`
- ☐ Echo `X-Trace-Id` from request header (or generate if missing)
- ☐ Log `trace_id` with every error for correlation
- ☐ Include status code in both HTTP response and JSON body
- ☐ FastAPI exception handlers catch all errors and format consistently

****Discussion:**** Save these `.http` snippets. Run them against your FastAPI server to verify error handling before submitting EX1.

Prepare Schemathesis for contract testing

Schemathesis acts as an adversarial client that tries to break your server's contract:

```
```bash
uv add schemathesis
```

**Motivation:** In EX1, you'll define OpenAPI schemas for `/movies` endpoints. Schemathesis generates edge-case requests (empty strings, negative IDs, oversized payloads) that stress-test your server's validation.

### Demo (optional if time permits):

```
Once EX1 is running, you'll run:
uv run schemathesis run http://localhost:8000/openapi.json \
--checks status_code_conformance \
--hypothesis-max-examples=50
```

Reinforce: This is optional for baseline EX1 but required for excellence submissions. Start with manual `.http` client tests, add Schemathesis after CRUD works.

## Document and share findings

Have each student post in their Discord lab thread:

- One `.http` snippet from the error envelope test cases they want to implement first
- One edge case they discovered (e.g., negative movie ID, missing trace header)
- One question about client-server-gateway architecture

## Before You Leave – Quick Verification

Run these commands to confirm your setup:

```
1. Client works
uv run python -m app.cli echo --city Haifa
```

```
2. Tests pass
uv run pytest -q tests/test_ping.py

3. REST Client installed
code --list-extensions | grep humao.rest-client

4. Error contract document exists
cat docs/contracts/http-errors.md | head -n 5
```

If the `code` CLI isn't on your PATH, open VS Code → Extensions and confirm **REST Client** is installed manually, then return to the terminal for the other checks.

**Post a screenshot of all four succeeding in your Discord thread.** If any fail, ask in `#helpdesk` before leaving.

## EX1 Briefing & Backlog (10 Minutes)

---

Link to full brief: [docs/exercises.md](https://docs/exercises.md)

### Core deliverables (due Monday, 30/03/2026):

- FastAPI server with CRUD for `/movies` and health endpoint
- SQLite/SQLModel persistence + seed/migration helpers (stretch for EX1; required before EX3)
- Error responses match `docs/contracts/http-errors.md`
- 80% branch coverage with pytest

### Backlog items (prioritize for excellence):

- Pagination & filtering ( `?limit=10&offset=20` )
- Rate limiting with `slowapi` (429 responses with `Retry-After` header)
- OpenAPI examples for happy/sad paths
- Schemathesis integration for fuzz testing
- ETag/ `If-None-Match` caching demo

**Architecture reminder:** Your Typer client from today becomes the test harness; FastAPI is the server. SQLite/SQLModel and Postgres sit beneath it, and an nginx gateway is optional once you add one. All components should propagate trace IDs.

**Next steps:** Implement the FastAPI **server** that responds to your Typer client. Build `/movies` CRUD endpoints, ensure errors match `docs/contracts/http-errors.md`, and add trace ID middleware. Bring questions about FastAPI routing and dependency injection.

## Common Pitfalls

---

- **Confusing client vs server responsibilities** – client initiates requests and handles responses; server listens, routes, and generates responses.
- **Forgetting gateway role** – gateways don't implement business logic, they route/transform/log requests between clients and servers.
- **Missing trace ID propagation** – every component (client, gateway, server) must accept and forward trace IDs for observability.
- **CRUD verb mix-ups** – using `POST /movies/{id}` for updates (should be PUT), or forgetting DELETE should return 204 even when item doesn't exist.

- **Leaking stack traces** – 500 errors must return sanitized error envelope, not raw exceptions.
- **Typer command exits immediately** – ensure `if __name__ == "__main__":` block calls `main()`.

## Troubleshooting Notes

---

- **nginx container won't start:** check `docker logs demo-gateway` for config syntax errors.
- **httpx.ConnectTimeout:** verify campus VPN/proxy settings; demonstrate adding longer timeout.
- **VS Code REST Client 401s:** some APIs require specific headers—check extension docs for auth setup.

## Student Success Criteria

---

By the end of Session 02, every student should be able to:

- ☐ Explain client-server-gateway architecture and draw a request/response flow diagram.
- ☐ Map CRUD verbs to the `/movies` API and choose the correct HTTP method for each action.
- ☐ Run nginx as a gateway and trace an HTTP request through docker logs.
- ☐ Build an HTTP client with httpx that generates trace IDs and validates responses.
- ☐ Capture `.http` request patterns and document server-side error contracts for EX1.

**If a student cannot check any box above, schedule an office-hours pairing soon.**

## AI Prompt Seeds

---

- "Explain the difference between HTTP client responsibilities and server responsibilities with code examples."
- "Generate nginx config that acts as a gateway: adds X-Trace-Id header and forwards to FastAPI backend."
- "Draft a Typer CLI that wraps httpx requests and propagates trace IDs for observability."

# Session 03 – FastAPI Fundamentals (Movie Service v0)

---

docs/sessions/session-03.md

- **Date:** Monday, 16/03/2026
- **Theme:** Build a complete FastAPI movie service with dependency injection, validation, and automated tests.

## Learning Objectives

---

By the end of this session, you will:

- Build REST API endpoints with FastAPI using proper HTTP methods and status codes
- Validate request/response data with Pydantic v2 models and custom validators
- Implement dependency injection to manage shared resources (settings, repositories)
- Write automated tests using pytest and TestClient
- Run the same application locally and in Docker containers

## What You'll Build

---

A working FastAPI backend that:

- Stores movies in memory with CRUD operations
- Validates all incoming data (year ranges, required fields)
- Returns proper HTTP status codes (200, 201, 404, 422)
- Includes comprehensive test coverage
- Runs identically on your machine and in Docker

## EX1 Baseline (In-Memory Only)

- This session is **the EX1 deliverable**: students leave class with a FastAPI backend, pytest suite, Docker packaging, and REST client scripts that satisfy the HTTP API portion of the exercise—even if data only lives in memory for now. Think of it as microservice #1 in the eventual EX3 stack.
- In-memory storage is intentional at this checkpoint; it keeps the focus on HTTP verbs, validation, dependency injection, and smoke testing. Everyone can demo the service (curl, REST Client, Docker) without juggling SQL yet.
- Keep the architecture clean so future persistence or UI layers can drop in without changing HTTP routes.
- Encourage students to push this code to their EX1 repo immediately; later DB hardening layers on top of the same contract rather than creating a new app.
- **Architecture decision:** Use `MovieBase` + `Movie` + `MovieCreate` from day one; when you introduce `SQLModel`, add `table=True` and optionally `MovieRead` for response serialization while keeping the HTTP contract identical.
- **Interfaces stay stable on purpose:** The `repository` interface and dependency wiring you build now make storage swaps (dict → SQLite → Postgres) a drop-in change. Keep the boundaries clean and small.

## Live Build Strategy (No Pre-Solved Example)

---

- This session is the canonical FastAPI walkthrough for the whole course, so we now build the example entirely from the scripted steps below instead of keeping a separate `examples/fastapi-calculator/` folder.
- Treat this document as the live-coding script: copy/paste the code fences when time is tight, or narrate each block while students type. Everything here feeds directly into Sessions 04–07, so keeping it inline prevents drift.

- Want a rehearse-ready reference? Pair these steps with the REST Client snippets in `examples.http` —they hit the exact endpoints you create in Lab 1.
- Optional deep dives that truly need their own repos (for example the blockchain mini-demo) are cataloged in `examples/README.md` ; they stay disconnected from the core FastAPI storyline on purpose.

## Prerequisites

---

Before class, complete these setup steps:

### 1. Install Python 3.12 with uv:

```
Install uv package manager
curl -LsSf https://astral.sh/uv/install.sh | sh

Restart your shell to load uv
exec "$SHELL" -l

Verify uv installed correctly
uv --version

Install Python 3.12
uv python install 3.12
```

### 2. Create project workspace:

```
mkdir hello-uv && cd hello-uv
git init
```

Create `.gitignore` to prevent committing sensitive/generated files:

```
cat > .gitignore << 'EOF'
Python
__pycache__/
*.py[cod]
*$py.class
.pytest_cache/

Virtual environment
.venv/

Environment variables (contains secrets)
.env

IDEs
.vscode/
.idea/

OS
.DS_Store
EOF
```

### 3. Initialize uv project and add dependencies:

```
Initialize creates pyproject.toml
uv init --no-readme

Add runtime dependencies
```

```
uv add fastapi uvicorn pydantic pydantic-settings httpx

Add dev/test dependencies
uv add --dev pytest
```

### What just happened:

- `pyproject.toml` - Declares your project dependencies
- `uv.lock` - Locks exact versions for reproducible installs (commit this!)
- `.venv/` - Virtual environment with isolated packages (don't commit)
- Dev dependencies (pytest) live in their own group so Docker builds can skip them

Verify the files exist:

```
ls -la # Should see: .gitignore, pyproject.toml, uv.lock, .venv/
```

## 4. Create environment configuration:

Create `.env.example` (committed template):

```
cat > .env.example << 'EOF'
MOVIE_APP_NAME="Movie Service"
MOVIE_DEFAULT_PAGE_SIZE=20
MOVIE_FEATURE_PREVIEW=false
EOF
```

Copy to `.env` for local development (stays private):

```
cp .env.example .env
```

**Note:** `.env` contains your local overrides and should never be committed.

## 5. Verify setup works:

```
Check Python version in virtual environment
uv run python --version # Should show 3.12.x

Test that dependencies import correctly
uv run python -c "import fastapi, pydantic; print('Ready!')"

Verify project files
ls -la # Should see: .env, .env.example, .gitignore, pyproject.toml, uv.lock, .venv/
```

**If any command fails, stop and troubleshoot before continuing to class.**

## 6. Make initial commit:

```
git add .gitignore .env.example pyproject.toml uv.lock
git commit -m "Initial project setup with uv"
```

**Important:** We do NOT commit `.env` (secrets) or `.venv/` (too large).

## Toolkit Snapshot

- **FastAPI** – async-ready web framework that maps HTTP verbs to Python functions and autogenerates OpenAPI docs.
- **uv** – Python/packaging manager from Astral; manages Python 3.12 installs, virtual envs, and dependency locking.
- **Pydantic v2** – data validation library; enforces request/response shapes via type hints and `Field` metadata.
- **pytest + TestClient** – testing framework plus FastAPI's HTTP client for asserting status codes and payloads.
- **httpx** – modern HTTP library (sync/async) you can reuse for CLI checks or future integration tests.
- **Docker** – optional packaging step that proves the same FastAPI app runs outside your local Python setup.

## Session Agenda

Segment	Duration	Format	Focus
Part A – Recap & intent	5 min	Guided discussion	Surface wins from Session 02, confirm EX1 blockers, frame why FastAPI is today's focus.
Part A – FastAPI fundamentals	15 min	Talk + board	Map HTTP verbs to path operations, highlight what <code>/docs</code> gives you for free.
Part A – Request flow walkthrough	10 min	Diagram walkthrough	Trace HTTP → FastAPI → dependency injection → repository → response.
Part A – Live demo: docs + TestClient	15 min	Live coding	Show how FastAPI auto-generates docs/tests before we build it ourselves.
Break	10 min	—	Walk/stretch, reset terminals.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>Scaffold config, models, repository, and routes for Movie Service v0.</b>
Break	10 min	—	Hydrate, swap drivers.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided testing</b>	<b>Pytest fixtures, TestClient, regression hunts, red→green loops.</b>
Docker deployment demo	10 min	Live demo	Containerize the same app, compare local vs Docker runs.
Wrap-up & checklist	10 min	Discussion + Q&A	Review deliverables and open Q&A.

## Part A – Theory & Live Demos (45 minutes)

### Talk track:

- Kick off with a 60-second recap of Session 02's HTTP anatomy and connect it to today's goal: code the same contracts in FastAPI.
- Draw the request flow (client → FastAPI app → dependency injection → repository → response) before showing any code so students see where each file fits.
- Bounce between the board and the live server ( `/docs` , `pytest` ) so every concept immediately maps to feedback loops they'll use in Labs 1 and 2.

### 1. FastAPI Path Operations (15 min)

FastAPI maps HTTP methods to Python functions:

```

@app.get("/movies") # GET request - retrieve data
def list_movies(): ...

@app.post("/movies") # POST request - create new resource
def create_movie(): ...

@app.get("/movies/{id}") # GET with path parameter
def get_movie(id: int): ...

@app.delete("/movies/{id}") # DELETE request - remove resource
def delete_movie(id: int): ...

```

#### Status codes matter:

- 200 - Successful GET/PUT/DELETE
- 201 - Resource created (POST)
- 204 - Success with no content (DELETE)
- 404 - Resource not found
- 422 - Validation error

## 2. Pydantic Models for Validation (10 min)

Pydantic automatically validates request/response data:

```

from pydantic import BaseModel, Field

class MovieCreate(BaseModel):
 title: str
 year: int = Field(ge=1900, le=2100) # Between 1900-2100
 genre: str

```

Benefits:

- Automatic type checking
- Clear error messages
- Self-documenting API
- Auto-generated OpenAPI docs

## 3. Dependency Injection (10 min)

Share resources across endpoints without globals:

```

from fastapi import Depends

def get_repository():
 return MovieRepository()

@app.post("/movies")
def create_movie(
 payload: MovieCreate,
 repo: MovieRepository = Depends(get_repository)
):
 return repo.create(payload)

```

Why use DI:

- Easy to test (swap dependencies)
- No global state issues
- Clear dependencies in function signatures

#### 4. Local Feedback Loops (10 min)

- Run the API with `uv run uvicorn movie_service.app.main:app --reload` for instant reloads.
- Keep pytest green: `uv run pytest movie_service/tests -q`.
- Use `curl` or `httpie` to hit endpoints exactly as your automated tests do.
- The faster the loop, the easier it is to spot regressions before moving on.

## Part B – Lab 1: Build the Movie API (45 minutes)

**Goals:** Ship an in-memory FastAPI service with health + CRUD endpoints, dependency injection, and clean separation between config, models, repositories, and routes.

#### Facilitation tips:

- Keep VS Code and the terminal side by side; pause after each file so students catch up before moving on.
- Continuously connect each file to the earlier request-flow diagram so the mental model stays intact.
- Reinforce that clean boundaries make storage swaps straightforward without route changes.

Follow these steps to build a working FastAPI application from scratch.

### Step 1: Create Project Structure (5 min)

#### Verify you're in the project root:

```
pwd # Should show ../hello-uv
ls # Should see: .env, pyproject.toml, uv.lock
```

#### Create the movie\_service package:

```
Create directory structure
mkdir -p movie_service/{app,tests,scripts}

Create Python package markers (__init__.py makes directories importable)
touch movie_service/__init__.py
touch movie_service/app/__init__.py
touch movie_service/tests/__init__.py
touch movie_service/scripts/__init__.py
```

#### Verify structure:

```
If you have tree installed:
tree movie_service -L 2

Otherwise:
find movie_service -type f -name "*.py"
```

Your structure should look like:

```

hello-uv/ # ← You are here
├── .env # Local config (not committed)
├── .env.example # Template (committed)
├── .gitignore # What to ignore (committed)
├── pyproject.toml # Dependencies (committed)
├── uv.lock # Locked versions (committed)
├── .venv/ # Virtual env (not committed)
└── movie_service/
 ├── __init__.py
 ├── app/
 │ ├── __init__.py
 │ ├── config.py # ← We'll create these next
 │ ├── models.py
 │ ├── repository.py
 │ ├── dependencies.py
 │ └── main.py
 ├── tests/
 │ ├── __init__.py
 │ ├── conftest.py # ← Test fixtures
 │ └── test_movies.py # ← Test cases
 └── scripts/
 ├── __init__.py
 └── export_openapi.py

```

## Step 2: Configure Settings (5 min)

Create `movie_service/app/config.py` :

```

filepath: movie_service/app/config.py
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
 """Application settings loaded from environment variables."""
 app_name: str = "Movie Service"
 default_page_size: int = 20
 feature_preview: bool = False

 model_config = SettingsConfigDict(
 env_file=".env", # Load from .env file
 env_prefix="MOVIE_", # Only read MOVIE_* variables
 extra="ignore", # Ignore unknown variables
)

```

**What this does:**

- Loads configuration from `.env` file
- Reads variables starting with `MOVIE_` prefix
- Provides defaults if variables aren't set
- Type-checks all values

## Step 3: Define Pydantic Models (5 min)

Create `movie_service/app/models.py` :

```

filepath: movie_service/app/models.py
from __future__ import annotations

```

```

from pydantic import BaseModel, Field, model_validator

class MovieBase(BaseModel):
 """Shared fields for create/read models.

 Designed to stay unchanged when swapping persistence layers (for example,
 when moving to SQLModel). The split between MovieBase/Movie/MovieCreate
 establishes the pattern reused throughout the course.
 """
 title: str
 year: int = Field(ge=1900, le=2100)
 genre: str

class Movie(MovieBase):
 """Response model that includes the server-generated ID.

 When migrating to SQLModel, add `table=True` while keeping the HTTP
 contract identical.
 """
 id: int

class MovieCreate(MovieBase):
 """Incoming payload with validation + normalization.

 This validator carries forward unchanged when the persistence layer swaps,
 demonstrating how validation rules survive storage changes.
 """

 @model_validator(mode="after")
 def normalize_genre(self) -> "MovieCreate":
 """Title-case the genre: 'sci-fi' -> 'Sci-Fi'."""
 self.genre = self.genre.title()
 return self

```

### Why split models from controllers:

- Controllers (FastAPI routes) only care about IO contracts.
- Repositories reuse the same schemas without importing FastAPI.
- Persistence swaps can reuse the same models without route changes.

### Step 4: Build the Repository (10 min)

Create `movie_service/app/repository.py`:

```

filepath: movie_service/app/repository.py
from __future__ import annotations

from typing import Dict

from .models import Movie, MovieCreate

class MovieRepository:
 """In-memory storage for movies.

 You can swap in SQLModel + SQLite later without changing the
 interface (list/create/get/delete), so routes stay the same.
 """

```

```

def __init__(self) -> None:
 self._items: Dict[int, Movie] = {}
 self._next_id = 1

def list(self) -> list[Movie]:
 """Get all movies.

 Always returns list[Movie] for consistency. SQLAlchemy queries
 return sequences, but convert with list() to maintain this interface.
 """
 return list(self._items.values())

def create(self, payload: MovieCreate) -> Movie:
 """Add a new movie and return it with assigned ID.

 In a SQLAlchemy-backed version this maps to `session.add()` +
 `session.commit()` while keeping the same signature.
 """
 movie = Movie(id=self._next_id, **payload.model_dump())
 self._items[movie.id] = movie
 self._next_id += 1
 return movie

def get(self, movie_id: int) -> Movie | None:
 """Get a movie by ID, or None if not found.

 A SQLAlchemy-backed repository would use `session.get(Movie, movie_id)`.
 """
 return self._items.get(movie_id)

def delete(self, movie_id: int) -> None:
 """Remove a movie by ID.

 A SQLAlchemy-backed repository would call `session.delete()` +
 `session.commit()`.
 """
 self._items.pop(movie_id, None)

def clear(self) -> None:
 """Remove all movies (useful for tests).

 SQLAlchemy-backed fixtures should use separate test databases instead of
 clearing a shared repository.
 """
 self._items.clear()
 self._next_id = 1

```

### Key features:

- Repository depends only on the domain models.
- Controllers can stay lean—no storage details leak into FastAPI handlers.
- Tests can import `MovieRepository` without touching FastAPI.

## Step 5: Wire Up Dependencies (5 min)

Create `movie_service/app/dependencies.py`:

```

filepath: movie_service/app/dependencies.py
from collections.abc import Generator
from typing import Annotated

from fastapi import Depends

```

```

from .config import Settings
from .repository import MovieRepository

Create singletons
_settings = Settings()
_repository = MovieRepository()

def get_settings() -> Settings:
 """Provide settings to endpoints."""
 return _settings

def get_repository() -> Generator[MovieRepository, None, None]:
 """Provide repository to endpoints."""
 yield _repository

Type aliases for cleaner endpoint signatures
SettingsDep = Annotated[Settings, Depends(get_settings)]
RepositoryDep = Annotated[MovieRepository, Depends(get_repository)]

```

### Why this pattern:

- One instance of Settings shared across all requests
- One instance of Repository shared across all requests
- Easy to swap implementations for testing
- Type hints make code self-documenting

## Step 6: Build the FastAPI App (15 min)

Create `movie_service/app/main.py` :

```

filepath: movie_service/app/main.py
from __future__ import annotations

import logging

from fastapi import FastAPI, HTTPException, status

from .dependencies import RepositoryDep, SettingsDep
from .models import Movie, MovieCreate

logger = logging.getLogger("movie-service")
logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")

app = FastAPI(title="Movie Service", version="0.1.0")

@app.get("/health", tags=["diagnostics"])
def health(settings: SettingsDep) -> dict[str, str]:
 """Health check endpoint."""
 return {"status": "ok", "app": settings.app_name}

@app.get("/movies", response_model=list[Movie], tags=["movies"])
def list_movies(repository: RepositoryDep) -> list[Movie]:
 """Get all movies."""
 return list(repository.list())

```

```

@app.post(
 "/movies",
 response_model=Movie,
 status_code=status.HTTP_201_CREATED,
 tags=["movies"],
)
def create_movie(
 payload: MovieCreate,
 repository: RepositoryDep,
) -> Movie:
 """Create a new movie."""
 movie = repository.create(payload)
 logger.info("movie.created id=%s title=%s", movie.id, movie.title)
 return movie

@app.get("/movies/{movie_id}", response_model=Movie, tags=["movies"])
def read_movie(movie_id: int, repository: RepositoryDep) -> Movie:
 """Get a specific movie by ID."""
 movie = repository.get(movie_id)
 if movie is None:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail="Movie not found",
)
 return movie

@app.delete(
 "/movies/{movie_id}",
 status_code=status.HTTP_204_NO_CONTENT,
 tags=["movies"],
)
def delete_movie(movie_id: int, repository: RepositoryDep) -> None:
 """Delete a movie by ID."""
 if repository.get(movie_id) is None:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail="Movie not found",
)
 repository.delete(movie_id)
 logger.info("movie.deleted id=%s", movie_id)

```

**Controller responsibility stays thin:** FastAPI handlers import the Pydantic models for type safety but let the repository enforce storage rules and business logic.

## Step 7: Run the API (5 min)

**Verify you're in the correct directory:**

```

pwd # Should show ../hello-uv (NOT ../hello-uv/movie_service)
ls # Should see: movie_service/, pyproject.toml, .env

```

**Start the development server:**

```

From the hello-uv directory
uv run uvicorn movie_service.app.main:app --reload

```

**Understanding the command:**

- `uv run` - Runs command in the virtual environment (no activation needed)
- `--reload` - Auto-restarts server when code changes (development only, remove in production)

You'll see:

```
INFO: Will watch for changes in these directories: ['/path/to/hello-uv']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [12345] using WatchFiles
INFO: Started server process [12346]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

### Test it works:

1. **Interactive API docs:** Open `http://127.0.0.1:8000/docs` in your browser

- See all endpoints with request/response schemas
- Try creating a movie using the "Try it out" button
- Check the "Schemas" section to see your Pydantic models

2. **Test with curl:**

```
Health check
curl http://localhost:8000/health
Response: {"status":"ok","app":"Movie Service"}

Create a movie
curl -X POST http://localhost:8000/movies \
 -H "Content-Type: application/json" \
 -d '{"title": "Inception", "year": 2010, "genre": "sci-fi"}'
Response: {"id":1,"title":"Inception","year":2010,"genre":"Sci-Fi"}
Notice: genre normalized to "Sci-Fi"

List all movies
curl http://localhost:8000/movies
Response: [{"id":1,"title":"Inception","year":2010,"genre":"Sci-Fi"}]

Get specific movie
curl http://localhost:8000/movies/1
Response: {"id":1,"title":"Inception","year":2010,"genre":"Sci-Fi"}

Delete a movie
curl -X DELETE http://localhost:8000/movies/1
Response: (empty body with 204 status)

Verify deletion (should return 404)
curl http://localhost:8000/movies/1
Response: {"detail":"Movie not found"}
```

### Success criteria:

-  Server starts without errors
-  `/docs` page loads and shows all endpoints
-  Can create and retrieve movies via curl
-  Genre is normalized ("sci-fi" → "Sci-Fi")
-  Health endpoint returns your app name
-  404 returned for missing movies

### If something fails:

- Check you're in `hello-uv/` directory (not `movie_service/`)
- Verify all files were created: `ls movie_service/app/*.py`
- Look for Python errors in the terminal output
- Test imports: `uv run python -c "from movie_service.app.main import app; print('OK')"`

## Part C – Lab 2: Test Everything (45 minutes)

**Goals:** Turn pytest into a guardrail so every regression is caught before Docker or GitHub.

### Facilitation tips:

- Start by running a single failing test to show how fast TestClient feedback arrives.
- Narrate the purpose of each fixture (state isolation, dependency overrides) before writing assertions.
- Celebrate the red→green moment, then immediately demonstrate how a removed validator is caught.

### Step 1: Create Test Fixtures (10 min)

Create `movie_service/tests/conftest.py` :

```
filepath: movie_service/tests/conftest.py
import pytest
from fastapi.testclient import TestClient

from movie_service.app.main import app
from movie_service.app.dependencies import get_repository

@pytest.fixture(autouse=True)
def clear_repository():
 """Clear repository before and after each test."""
 repo = next(get_repository())
 repo.clear()
 yield
 repo.clear()

@pytest.fixture
def client():
 """Provide a TestClient for making requests."""
 return TestClient(app)
```

### What fixtures do:

- `clear_repository` - Prevents tests from affecting each other
- `client` - Lets you make HTTP requests to your app
- `autouse=True` - Runs automatically before every test

### Step 2: Write Comprehensive Tests (25 min)

Create `movie_service/tests/test_movies.py` :

```

filepath: movie_service/tests/test_movies.py
"""
Comprehensive test suite for Movie Service API.

Tests use the 'client' fixture from conftest.py, which provides
a TestClient for making HTTP requests to our FastAPI app without
needing a real server.
"""

Note: The 'client' fixture is automatically discovered from conftest.py
No imports needed here – pytest handles fixture injection

def test_health_includes_app_name(client):
 """Health endpoint returns status and app name."""
 response = client.get("/health")
 assert response.status_code == 200
 data = response.json()
 assert data["status"] == "ok"
 assert data["app"] == "Movie Service"

def test_create_movie_returns_201_and_payload(client):
 """Creating a movie returns 201 with normalized payload."""
 response = client.post(
 "/movies",
 json={"title": "The Matrix", "year": 1999, "genre": "sci-fi"},
)
 assert response.status_code == 201
 payload = response.json()
 assert payload["title"] == "The Matrix"
 assert payload["year"] == 1999
 assert payload["genre"] == "Sci-Fi" # normalized by validator
 assert payload["id"] == 1

def test_movie_ids_increment(client):
 """Repository assigns sequential IDs."""
 first = client.post(
 "/movies",
 json={"title": "Blade Runner", "year": 1982, "genre": "sci-fi"},
).json()["id"]
 second = client.post(
 "/movies",
 json={"title": "Blade Runner 2049", "year": 2017, "genre": "sci-fi"},
).json()["id"]
 assert second == first + 1

def test_list_movies_returns_empty_array_initially(client):
 """Empty repository returns empty array."""
 response = client.get("/movies")
 assert response.status_code == 200
 assert response.json() == []

def test_list_movies_returns_created_movie(client):
 """Can retrieve movies after creating them."""
 client.post(
 "/movies",
 json={"title": "Dune", "year": 2021, "genre": "sci-fi"},
)

 response = client.get("/movies")
 assert response.status_code == 200
 movies = response.json()

```

```

assert len(movies) == 1
assert movies[0]["title"] == "Dune"

def test_get_movie_by_id(client):
 """Can retrieve specific movie by ID."""
 create_response = client.post(
 "/movies",
 json={"title": "Arrival", "year": 2016, "genre": "sci-fi"},
)
 movie_id = create_response.json()["id"]

 response = client.get(f"/movies/{movie_id}")
 assert response.status_code == 200
 movie = response.json()
 assert movie["title"] == "Arrival"
 assert movie["id"] == movie_id

def test_get_missing_movie_returns_404(client):
 """Requesting non-existent movie returns 404."""
 response = client.get("/movies/9999")
 assert response.status_code == 404
 assert response.json()["detail"] == "Movie not found"

def test_delete_movie(client):
 """Can delete a movie and it's gone afterwards."""
 create_response = client.post(
 "/movies",
 json={"title": "Interstellar", "year": 2014, "genre": "sci-fi"},
)
 movie_id = create_response.json()["id"]

 response = client.delete(f"/movies/{movie_id}")
 assert response.status_code == 204

 get_response = client.get(f"/movies/{movie_id}")
 assert get_response.status_code == 404

def test_delete_missing_movie_returns_404(client):
 """Deleting non-existent movie returns 404."""
 response = client.delete("/movies/9999")
 assert response.status_code == 404

def test_create_movie_rejects_year_too_old(client):
 """Year before 1900 is rejected with 422."""
 response = client.post(
 "/movies",
 json={"title": "Metropolis", "year": 1800, "genre": "sci-fi"},
)
 assert response.status_code == 422
 detail = response.json()["detail"]
 assert any("year" in str(err).lower() for err in detail)

def test_create_movie_rejects_year_too_new(client):
 """Year after 2100 is rejected with 422."""
 response = client.post(
 "/movies",
 json={"title": "Future Film", "year": 2200, "genre": "sci-fi"},
)
 assert response.status_code == 422

```

```
def test_create_movie_rejects_missing_title(client):
 """Missing required field returns 422."""
 response = client.post(
 "/movies",
 json={"year": 2020, "genre": "drama"},
)
 assert response.status_code == 422

def test_create_movie_rejects_missing_genre(client):
 """Genre is required."""
 response = client.post(
 "/movies",
 json={"title": "Her", "year": 2013},
)
 assert response.status_code == 422
```

### Step 3: Run the Tests (5 min)

```
cd hello-uv
uv run pytest movie_service/tests -v
```

You should see:

```
test_health_includes_app_name PASSED
test_create_movie_returns_201_and_payload PASSED
test_movie_ids_increment PASSED
...
===== 13 passed in 0.45s =====
```

#### If tests fail:

1. Read the error message carefully
2. Check which assertion failed
3. Print the response: `print(response.json())`
4. Fix the code, rerun the test

### Step 4: Practice Red→Green Testing (5 min)

#### Live demonstration of test-driven development:

1. **Break something** - Comment out the genre validator in `movie_service/app/models.py` :

```
@model_validator(mode="after")
def normalize_genre(self) -> "MovieCreate":
"""Title-case the genre: 'sci-fi' -> 'Sci-Fi'."""
self.genre = self.genre.title()
return self
```

2. **Run the specific test** - See it fail:

```
uv run pytest movie_service/tests/test_movies.py::test_create_movie_returns_201_and_payload -v
```

#### Expected failure output:

```
FAILED test_create_movie_returns_201_and_payload
AssertionError: assert 'sci-fi' == 'Sci-Fi'
- Sci-Fi
+ sci-fi
```

This is the **RED** phase - test fails because behavior doesn't match expectations.

3. **Fix it** - Uncomment the validator code

4. **Run tests again** - See them pass:

```
uv run pytest movie_service/tests/test_movies.py::test_create_movie_returns_201_and_payload -v
```

**Expected success output:**

```
PASSED test_create_movie_returns_201_and_payload
```

This is the **GREEN** phase - test passes, behavior matches expectations.

**The lesson:** Tests catch regressions immediately. The Red→Green→Refactor cycle:

-  **Red:** Write a failing test (or break something intentionally)
-  **Green:** Make it pass with minimal code
-  **Refactor:** Clean up while keeping tests green

Always run tests before committing code to ensure you don't break existing functionality.

## Part D – Docker Deployment (10 minutes)

Run the exact same application in a container with identical behavior.

### Create Dockerfile

Create `movie_service/Dockerfile`:

```
filepath: movie_service/Dockerfile
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
 PYTHONUNBUFFERED=1 \
 PATH="/app/.venv/bin:$PATH"

WORKDIR /app

Copy uv binary from official image (faster than pip install)
COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv

Copy dependency files first (better layer caching)
COPY pyproject.toml uv.lock ./

Install dependencies using locked versions
--frozen: don't update uv.lock, use exact versions
--no-dev: skip pytest and other dev dependencies
RUN uv sync --frozen --no-dev
```

```
Copy application code
COPY movie_service ./movie_service

Copy env template (container will use these defaults)
COPY .env.example ./env

EXPOSE 8000

CMD ["uv", "run", "uvicorn", "movie_service.app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

### What this Dockerfile does:

- Uses Python 3.12 slim base (smaller image size)
- Copies uv binary from official image (consistent tooling)
- Installs exact versions from `uv.lock` (reproducible builds)
- `--frozen` prevents dependency resolution (faster, deterministic)
- `--no-dev` skips test dependencies (smaller production image)
- Copies `.env.example` as `.env` for container defaults

### Build and Run

```
Ensure you're in hello-uv directory
pwd # Should show ../hello-uv

Build the image
docker build -t movie-service -f movie_service/Dockerfile .

Run the container
docker run --rm -p 8000:8000 --name movie-service movie-service
```

### Test the containerized app:

```
In a new terminal:
curl http://localhost:8000/health
curl http://localhost:8000/movies

Or visit http://127.0.0.1:8000/docs in your browser
```

### Stop the container:

```
Press Ctrl+C in the docker run terminal, OR:
docker stop movie-service
```

### Key insights:

- Docker ensures identical behavior across environments (dev, staging, production)
- `uv.lock` guarantees same dependency versions everywhere
- Container includes only production dependencies (smaller, faster, more secure)
- No "works on my machine" problems

### Compare local vs Docker:

Aspect	Local	Docker
Startup	<code>uv run uvicorn ...</code>	<code>docker run ...</code>
Dependencies	From local <code>.venv</code>	Baked into image
Port	127.0.0.1:8000	0.0.0.0:8000
Data persistence	In-memory (both)	In-memory (both)
Environment	From <code>.env</code>	From <code>.env</code> in image

## Part E – Export the API Contract

Document your API with machine-readable OpenAPI schema.

Create `movie_service/scripts/export_openapi.py` :

```
filepath: movie_service/scripts/export_openapi.py
"""
Export OpenAPI schema for Movie Service.

Generates a JSON schema file that can be used for:
- Client code generation (openapi-generator, swagger-codegen)
- API documentation (swagger-ui, redoc)
- Contract testing (schemathesis, dredd)
- Validation and monitoring
"""
import json
from pathlib import Path

from movie_service.app.main import app

Generate OpenAPI schema from FastAPI app
schema = app.openapi()

Create output directory (inside movie_service for cleaner structure)
contracts_dir = Path("movie_service/contracts")
contracts_dir.mkdir(parents=True, exist_ok=True)

Write schema to file
output_path = contracts_dir / "openapi.json"
output_path.write_text(json.dumps(schema, indent=2))

Print summary
print(f"Exported OpenAPI schema to {output_path}")
print(f" Title: {schema['info']['title']}")
print(f" Version: {schema['info']['version']}")
print(f" Endpoints: {len(schema['paths'])}")
print(f"\nUse this file to:")
print(f" - Generate client SDKs (Python, TypeScript, Java, etc.)")
print(f" - Validate API responses in tests")
print(f" - Generate interactive documentation")
print(f" - Monitor API contract compliance")
```

Run it:

```
uv run python -m movie_service.scripts.export_openapi
```

Expected output:

```
Exported OpenAPI schema to movie_service/contracts/openapi.json
Title: Movie Service
Version: 0.1.0
Endpoints: 5
```

#### What this gives you:

- **Machine-readable API contract** - Can be validated automatically
- **Client code generation** - Generate SDKs for any language
- **Documentation** - Always in sync with actual code
- **Contract testing** - Verify responses match schema
- **Living documentation** - Never gets outdated

#### View the schema:

```
cat movie_service/contracts/openapi.json | head -30
```

## Wrap-Up & Deliverables

---

### What You Built

- FastAPI application with CRUD operations
- Pydantic validation with custom validators
- Dependency injection for settings and repositories
- Comprehensive test suite with 13 passing tests
- Docker containerization
- OpenAPI schema export

### Complete Checklist

Before moving forward, ensure:

- ☐ All tests pass: `uv run pytest movie_service/tests -v`
- ☐ Server runs locally: `uv run uvicorn movie_service.app.main:app --reload`
- ☐ Docker build succeeds: `docker build -t movie-service -f movie_service/Dockerfile .`
- ☐ API docs load at `/docs`
- ☐ Can create, list, get, and delete movies
- ☐ Validation errors return 422 with details
- ☐ OpenAPI schema exports successfully
- ☐ Code is committed and tagged `session-03-complete` for easy rollback

### Next Steps

#### Enhancements to add:

1. `PUT /movies/{id}` endpoint for updates
2. Pagination with `skip` and `limit` parameters
3. Search/filter endpoints

4. More comprehensive error handling
5. Update README with setup instructions

**Why the repository pattern matters:** The `list/create/get/delete` interface stays identical by design. Only the *implementation* changes when you swap storage engines, which keeps HTTP routes stable as the stack matures.

## Troubleshooting

---

### Quick Diagnostics

If anything isn't working, run these checks first:

```
1. Verify you're in the right directory
pwd # Should show ../hello-uv
ls # Should see: movie_service/, pyproject.toml, .env, .gitignore

2. Check uv is installed
uv --version # Should show uv version number

3. Verify Python version
uv run python --version # Should show 3.12.x

4. Test dependencies import
uv run python -c "import fastapi; print('FastAPI OK!')"

5. Test module imports
uv run python -c "from movie_service.app.main import app; print('Imports OK!')"
```

---

### "uv: command not found"

**Fix:**

```
Install uv
curl -LsSf https://astral.sh/uv/install.sh | sh

Restart shell to load PATH
exec "$SHELL" -l

Verify
uv --version
```

---

### "uv.lock is out of date"

**Fix:**

```
Regenerate lock file
uv lock

Or force sync
uv sync --refresh
```

---

## "No module named 'movie\_service'"

### Causes:

1. Running from wrong directory
2. Missing `__init__.py` files
3. Typo in import path

### Fix:

```
1. Ensure you're in hello-uv/ (not movie_service/)
cd "$(git rev-parse --show-toplevel)" 2>/dev/null || cd /path/to/hello-uv
pwd # Should show ../hello-uv (NOT ../hello-uv/movie_service)

2. Verify package markers exist
ls movie_service/__init__.py
ls movie_service/app/__init__.py
ls movie_service/tests/__init__.py
ls movie_service/scripts/__init__.py

3. Test import
uv run python -c "import movie_service; print('OK')"
```

---

## Server won't start

```
Check dependencies installed
uv run python -c "import fastapi; print('FastAPI OK')"
```

```
Check for port conflicts (something else using 8000)
lsof -i :8000
If something is there: kill -9 <PID>

Try a different port
uv run uvicorn movie_service.app.main:app --port 8001
```

---

## Tests failing

```
Run single test with verbose output
uv run pytest movie_service/tests/test_movies.py::test_health_includes_app_name -vv

Add debug output in test
def test_something(client):
 response = client.post("/movies", json={...})
 print(response.status_code) # Add this
 print(response.json()) # Add this
 assert ...

Run with print output visible
uv run pytest movie_service/tests -v -s
```

## Docker build fails

```
Build with verbose output
docker build --progress=plain -t movie-service -f movie_service/Dockerfile .

Check required files exist
ls pyproject.toml uv.lock movie_service/app/main.py

Common issue: missing uv.lock
Fix: run `uv lock` to generate it

Test Docker locally before building
docker run --rm python:3.12-slim python --version
```

---

## Docker container exits immediately

```
Check container logs
docker logs movie-service

Run interactively to see errors
docker run --rm -it -p 8000:8000 movie-service sh

Inside container:
ls -la
python --version
python -m movie_service.app.main # Should fail with helpful error
```

---

## Port already in use

```
Find what's using port 8000
lsof -i :8000

Kill the process
kill -9 <PID>

Or use a different port
uv run uvicorn movie_service.app.main:app --port 8001
```

---

## "Ready!" verification fails

**Problem:** `import fastapi, pydantic` fails

**Fix:**

```
Reinstall dependencies
uv sync

Check what's installed
uv pip list

Verify pyproject.toml has dependencies
cat pyproject.toml
```

```
Try installing explicitly
uv add fastapi pydantic
```

## Key Takeaways

---

1. **FastAPI is productive:** Routes, validation, and docs are automatic
2. **Pydantic validates everything:** Catch errors before they cause problems
3. **Dependency injection is powerful:** Easy to test, easy to swap implementations
4. **Tests give confidence:** Run them often, commit when green
5. **Docker provides consistency:** Same behavior everywhere
6. **OpenAPI is free documentation:** Always up-to-date with your code

## Success Criteria

---

You're ready to move on when you can:

- ☒ Explain how a request flows through FastAPI (middleware → route → dependencies → handler)
- ☒ Create new endpoints with proper HTTP methods and status codes
- ☒ Add Pydantic validation rules and custom validators
- ☒ Write tests that cover happy paths and error cases
- ☒ Use dependency injection to share resources
- ☒ Run the same code locally and in Docker
- ☒ Export and understand the OpenAPI schema

**Schedule a mentor session if any box remains unchecked.**

## AI Prompt Seeds

---

- "Given this spec for `/movies` CRUD plus `/health`, scaffold config, models, repository, dependencies, and FastAPI routes that use `pydantic-settings` + dependency injection."
- "Write pytest cases (using FastAPI's `TestClient`) that prove movie creation, lookup, deletion, and divide-by-zero-style errors work; include a fixture that clears the in-memory repository between tests."
- "Propose a Dockerfile and uv commands so the exact same FastAPI app runs locally and in a container; explain why each instruction exists."
- "Summarize a verification checklist for students building this example live: which commands should they run after each lab block to ensure parity with the instructor?"

# Session 04 – Persisting the Movie Service (SQLite + SQLAlchemy)

---

docs/sessions/session-04.md

- **Date:** Monday, 23/03/2026
- **Theme:** Replace the in-memory repository with SQLite + SQLAlchemy, wire FastAPI to real sessions, and back everything with migrations + tests.

## Session Story

---

Session 03 shipped the Movie Service with in-memory storage. Session 04 keeps the same HTTP contract but persists to SQLite through SQLAlchemy, proving the repository abstraction, adding Alembic migrations, and hardening pytest fixtures so every test runs in a throwaway database. Students leave with durable data, migrations, and a repeatable seed script they can trust in later sessions.

## Instructor Notes

---

- `uv run pytest movie_service/tests -v` (baseline still green)
- Ensure `.gitignore` ignores `data/` but keeps Alembic revisions tracked (do not ignore `migrations/versions/` ; keep `migrations/env.py` and `migrations/script.py.mako` in version control)
- Generate the initial Alembic revision before running `init_db()` ; if `data/movies.db` already exists, delete it so alembic revision `--autogenerate` emits the `create_table`
- `uv run alembic upgrade head` and confirm `data/movies.db` timestamps change (this creates the SQLite file)
- `uv run python -m movie_service/scripts/seed_db` twice to verify idempotent seeding
- Smoke `uv run uvicorn movie_service.app.main:app --reload` then `curl http://localhost:8000/health` and `curl http://localhost:8000/movies`
- Ensure the `data/` folder is writable on classroom machines

## Learning Objectives

---

- Model movie data with SQLAlchemy tables while keeping request/response schemas intact.
- Configure a SQLite engine and FastAPI dependency that hands out scoped SQLAlchemy sessions.
- Replace the dict-backed repository with SQLAlchemy CRUD without changing the routes.
- Run pytest against isolated SQLite files to prevent cross-test contamination.
- Capture schema history with Alembic and seed working data via uv scripts.

## Deliverables (What You'll Build)

---

- `movie_service/app/models.py` SQLAlchemy definitions (`Movie` , `MovieCreate` , `MovieRead` , `MovieUpdate` ).
- `movie_service/app/database.py` with `engine` , `get_session` , and `init_db()` helpers.
- `movie_service/app/repository_db.py` backed by SQLAlchemy sessions for CRUD.
- FastAPI routes that still satisfy the Session 03 HTTP contract while persisting rows.
- Alembic scaffolding + initial migration and `movie_service/scripts/seed_db.py` .
- pytest fixtures that create/destroy temp SQLite databases per test file.

## Toolkit Snapshot

---

- **SQLModel** – Pydantic + SQLAlchemy hybrid that keeps request/response models close to table definitions.
- **SQLite** – file-based relational database that requires zero services for local dev.
- **SQLAlchemy/Alembic** – connection management + migrations.
- **pytest** – hermetic tests with dependency overrides.
- **uv** – dependency manager + runner for scripts, migrations, and uvicorn.

## Before Class (JiTT)

---

0. **Workflow reminder:** copy `docs/workflows/ai-assisted/templates/feature-brief.md` for each slice (settings, database, models, repository, tests, Alembic/seed). Keep each change under the 150-LOC guardrail in `docs/workflows/ai-assisted/README.md`.
1. Baseline: from the Session 03 code, run `uv run pytest movie_service/tests -v` and fix any failures. Commit/tag as `session-03-complete` so you can roll back easily.
2. Install the new dependencies inside `hello-uv`:

```
uv add sqlmodel alembic
```

3. Create a writable data directory and ignore it:

```
mkdir -p data
grep -qx 'data/' .gitignore || printf "\n# SQLite data\ndata\n" >> .gitignore
```

4. Extend `.env.example` and `.env` (relative path is fine for local dev):

```
MOVIE_DATABASE_URL="sqlite:///./data/movies.db"
```

5. Smoke-test imports:

```
uv run python - <<'PY'
import sqlmodel, alembic # noqa: F401
print("SQLModel + Alembic ready")
PY
```

## Migration Quickstart (from Session 03)

---

- Start clean: `uv run pytest movie_service/tests -v` and tag/branch `session-03-complete`.
- Add deps + data dir + env: `uv add sqlmodel alembic`; ensure `data/` exists/ignored and `.env(.example)` has `MOVIE_DATABASE_URL="sqlite:///./data/movies.db"`.
- Swap storage: follow Lab 1 to add `database.py`, update `config.py`, define SQLModel models, drop in `repository_db.py`, and wire `dependencies.py` + `main.py` (keep the HTTP contract identical).
- Tests: use the DB-aware fixtures in Lab 2 Step 1 and rerun the existing `movie_service/tests/test_movies.py`.
- Migrations: `uv run alembic init migrations`, edit `alembic.ini` + `migrations/env.py`, delete any pre-created `data/movies.db`, then `uv run alembic revision --autogenerate -m "create movies"` and `uv run alembic upgrade head`.
- Seeds: run `uv run python -m movie_service.scripts.seed_db` twice to confirm idempotency.
- Smoke: `uv run uvicorn movie_service.app.main:app --reload`, then `curl /health` and `/movies`.
- Commit/tag when green to mark the Session 04 baseline.

## Session Agenda

Time	Activity	Focus
10 min	Recap & intent	Why the repo abstraction makes the DB swap easy.
20 min	Data modeling primer	SQLModel basics, table vs response models, migrations.
45 min	<b>Lab 1</b>	<b>Database wiring + CRUD rewrite.</b>
10 min	Break	Encourage quick SQLite browser checks (e.g., DBeaver/DB Browser).
45 min	<b>Lab 2</b>	<b>Tests, Alembic, seed scripts.</b>
10 min	Wrap-up	Checklist + Q&A.

## Lab 1 – Persist CRUD with SQLModel (45 min)

Goal: move the repository + routes from in-memory storage to SQLite without touching the FastAPI contract.

Stay disciplined: one brief per subsystem (config, database, models, repository, routes). Generate → review → split before moving on so diffs stay small.

### Step 1: Configure Settings

movie\_service/app/config.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
 app_name: str = "Movie Service"
 default_page_size: int = 20
 feature_preview: bool = False
 database_url: str = "sqlite:///./data/movies.db"

 model_config = SettingsConfigDict(
 env_file=".env",
 env_prefix="MOVIE_",
 extra="ignore",
)
```

### Step 2: Create the database module

movie\_service/app/database.py

```
from collections.abc import Generator
from typing import Annotated

from fastapi import Depends
from sqlmodel import Session, SQLModel, create_engine

from .config import Settings

def get_settings() -> Settings:
 """Dependency for accessing application settings."""
 return Settings()
```

```

SettingsDep = Annotated[Settings, Depends(get_settings)]

Create engine using settings
settings = get_settings()
engine = create_engine(
 settings.database_url,
 echo=False,
 connect_args={"check_same_thread": False} if settings.database_url.startswith("sqlite") else {},
)

def init_db() -> None:
 """Initialize database tables. Import models before calling this."""
 from . import models # noqa: F401

 SQLAlchemyModel.metadata.create_all(engine)

def get_session() -> Generator[Session, None, None]:
 """Provide a database session for dependency injection."""
 with Session(engine) as session:
 yield session

SessionDep = Annotated[Session, Depends(get_session)]

```

Run the helper once so the database file exists (run from the project root). If you call this before creating the initial Alembic revision, delete `data/movies.db` before `alembic revision --autogenerate` so the migration emits the table:

```

uv run python - <<'PY'
from movie_service.app.database import init_db
init_db()
print("Created data/movies.db")
PY

```

### Step 3: Define SQLAlchemy classes

`movie_service/app/models.py`

```

from typing import Optional

from pydantic import field_validator
from sqlalchemy import Field, SQLAlchemyModel

class MovieBase(SQLAlchemyModel):
 """Base model with shared movie fields."""

 title: str = Field(min_length=1, max_length=200)
 year: int = Field(ge=1888, le=2100) # 1888 = first film ever made
 genre: str = Field(min_length=1, max_length=50)

class Movie(MovieBase, table=True):
 """Database model for movies."""

 __tablename__ = "movies"

 id: Optional[int] = Field(default=None, primary_key=True)

```

```

class MovieCreate(MovieBase):
 """Schema for creating a new movie."""

 @field_validator("genre")
 @classmethod
 def normalize_genre(cls, v: str) -> str:
 """Normalize genre to title case."""
 return v.strip().title()

 @field_validator("title")
 @classmethod
 def normalize_title(cls, v: str) -> str:
 """Strip whitespace from title."""
 return v.strip()

class MovieRead(MovieBase):
 """Schema for reading a movie."""

 id: int

class MovieUpdate(SQLModel):
 """Schema for updating a movie (all fields optional)."""

 title: Optional[str] = Field(default=None, min_length=1, max_length=200)
 year: Optional[int] = Field(default=None, ge=1888, le=2100)
 genre: Optional[str] = Field(default=None, min_length=1, max_length=50)

 @field_validator("genre")
 @classmethod
 def normalize_genre(cls, v: Optional[str]) -> Optional[str]:
 """Normalize genre to title case if provided."""
 return v.strip().title() if v else None

 @field_validator("title")
 @classmethod
 def normalize_title(cls, v: Optional[str]) -> Optional[str]:
 """Strip whitespace from title if provided."""
 return v.strip() if v else None

```

`MovieUpdate` is ready for a future `PUT/PATCH` route; the contract today stays `list/create/get/delete`.

## Step 4: Database-backed repository

`movie_service/app/repository_db.py`

```

from typing import Optional

from sqlmodel import Session, select

from .models import Movie, MovieCreate

class MovieRepository:
 """SQLite-backed storage for movies with proper session handling."""

 def __init__(self, session: Session) -> None:
 self.session = session

 def list(self, *, skip: int = 0, limit: int = 100) -> list[Movie]:
 """List all movies with pagination support."""

```

```

 statement = select(Movie).offset(skip).limit(limit)
 return list(self.session.exec(statement).all())

 def create(self, payload: MovieCreate) -> Movie:
 """Create a new movie and persist to database."""
 record = Movie.model_validate(payload)
 self.session.add(record)
 self.session.commit()
 self.session.refresh(record)
 return record

 def get(self, movie_id: int) -> Optional[Movie]:
 """Retrieve a movie by ID."""
 return self.session.get(Movie, movie_id)

 def delete(self, movie_id: int) -> bool:
 """Delete a movie by ID. Returns True if deleted, False if not found."""
 record = self.get(movie_id)
 if record is None:
 return False
 self.session.delete(record)
 self.session.commit()
 return True

 def delete_all(self) -> int:
 """Delete all movies and return count. Use with caution."""
 statement = select(Movie)
 records = self.session.exec(statement).all()
 count = len(records)
 for record in records:
 self.session.delete(record)
 self.session.commit()
 return count

```

## Step 5: Dependency wiring + routes

movie\_service/app/dependencies.py

```

from typing import Annotated

from fastapi import Depends

from .database import SessionDep
from .repository_db import MovieRepository

def get_repository(session: SessionDep) -> MovieRepository:
 """Dependency for accessing the movie repository."""
 return MovieRepository(session)

RepositoryDep = Annotated[MovieRepository, Depends(get_repository)]

```

movie\_service/app/main.py

```

from fastapi import FastAPI, HTTPException, status

from .database import SettingsDep
from .dependencies import RepositoryDep
from .models import MovieCreate, MovieRead

app = FastAPI(

```

```

 title="Movie Service",
 version="0.2.0",
 description="A service for managing movie data with SQLite persistence",
)

@app.get("/health", tags=["diagnostics"])
def health(settings: SettingsDep) -> dict[str, str]:
 """Health check endpoint."""
 return {"status": "ok", "app": settings.app_name}

@app.get("/movies", response_model=list[MovieRead], tags=["movies"])
def list_movies(
 repository: RepositoryDep,
 skip: int = 0,
 limit: int = 100,
) -> list[MovieRead]:
 """List all movies with optional pagination."""
 return repository.list(skip=skip, limit=limit)

@app.post(
 "/movies",
 response_model=MovieRead,
 status_code=status.HTTP_201_CREATED,
 tags=["movies"],
)
def create_movie(payload: MovieCreate, repository: RepositoryDep) -> MovieRead:
 """Create a new movie."""
 return repository.create(payload)

@app.get("/movies/{movie_id}", response_model=MovieRead, tags=["movies"])
def read_movie(movie_id: int, repository: RepositoryDep) -> MovieRead:
 """Retrieve a specific movie by ID."""
 movie = repository.get(movie_id)
 if movie is None:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Movie with id {movie_id} not found",
)
 return movie

@app.delete("/movies/{movie_id}", status_code=status.HTTP_204_NO_CONTENT, tags=["movies"])
def delete_movie(movie_id: int, repository: RepositoryDep) -> None:
 """Delete a specific movie by ID."""
 deleted = repository.delete(movie_id)
 if not deleted:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Movie with id {movie_id} not found",
)

```

## Step 6: Manual smoke test

```

uv run uvicorn movie_service.app.main:app --reload
curl -X POST http://localhost:8000/movies \
 -H "Content-Type: application/json" \
 -d '{"title": "Inception", "year": 2010, "genre": "sci-fi"}'
curl http://localhost:8000/movies

```

Run after `init_db()` or `uv run alembic upgrade head`; records should persist in `data/movies.db` across restarts.

## Lab 2 – Tests, Migrations, Seeds (45 min)

Goal: prove the database-backed service is safe to refactor by adding hermetic pytest fixtures, an Alembic migration, and a repeatable seed script.

Keep the briefs flowing: fixtures → Alembic → seed script. Commit between each chunk to avoid giant diffs.

### Step 1: Database-aware pytest fixtures

`movie_service/tests/conftest.py`

```
from collections.abc import Generator

import pytest
from fastapi.testclient import TestClient
from sqlmodel import Session, SQLModel, create_engine

from movie_service.app import models # noqa: F401
from movie_service.app.database import get_session
from movie_service.app.main import app

@pytest.fixture(name="engine")
def engine_fixture(tmp_path):
 """Create a temporary SQLite database for testing."""
 test_db = tmp_path / "test.db"
 test_engine = create_engine(
 f"sqlite:/// {test_db}",
 connect_args={"check_same_thread": False},
 echo=False,
)
 SQLModel.metadata.create_all(test_engine)
 yield test_engine
 # Cleanup: drop all tables and dispose engine
 SQLModel.metadata.drop_all(test_engine)
 test_engine.dispose()

@pytest.fixture(name="session")
def session_fixture(engine) -> Generator[Session, None, None]:
 """Provide a SQLModel session connected to the test database."""
 with Session(engine) as session:
 yield session
 # Rollback any uncommitted changes
 session.rollback()

@pytest.fixture(name="client")
def client_fixture(session: Session) -> Generator[TestClient, None, None]:
 """Provide a test client with overridden dependencies."""

 def get_session_override() -> Generator[Session, None, None]:
 yield session

 app.dependency_overrides[get_session] = get_session_override

 with TestClient(app) as test_client:
 yield test_client
```

```
Cleanup: remove override
app.dependency_overrides.clear()
```

## Step 2: Reuse Session 03 tests

Keep the assertions in `movie_service/tests/test_movies.py` ; the fixtures already isolate each test file. Run them:

```
uv run pytest movie_service/tests -v
```

## Step 3: Alembic workflow

Run this step before using `init_db()` or any command that creates `data/movies.db` . If you already created the file, delete it so `alembic revision --autogenerate` emits the `create_table` .

1. Scaffold Alembic:

```
uv run alembic init migrations
```

2. Keep revision files tracked (do **not** ignore `migrations/versions/` ; commit generated files so others can run upgrades).
3. Update `alembic.ini` to remove the hard-coded `sqlalchemy.url` (env.py will set it). Leave `script_location = migrations` .
4. Edit `migrations/env.py` :

```
filepath: migrations/env.py
from logging.config import fileConfig

from alembic import context
from sqlalchemy import engine_from_config, pool
from sqlmodel import SQLModel

Import all models to ensure they're registered with SQLModel metadata
from movie_service.app import models # noqa: F401
from movie_service.app.config import Settings

config = context.config

Get database URL from settings
settings = Settings()
if not config.get_main_option("sqlalchemy.url"):
 config.set_main_option("sqlalchemy.url", settings.database_url)

Interpret the config file for Python logging
if config.config_file_name is not None:
 fileConfig(config.config_file_name)

target_metadata = SQLModel.metadata

def run_migrations_offline() -> None:
 """Run migrations in 'offline' mode."""
 url = config.get_main_option("sqlalchemy.url")
 context.configure(
 url=url,
 target_metadata=target_metadata,
 literal_binds=True,
```

```

 dialect_opts={"paramstyle": "named"},
 render_as_batch=True, # Required for SQLite ALTER TABLE
)

 with context.begin_transaction():
 context.run_migrations()

def run_migrations_online() -> None:
 """Run migrations in 'online' mode."""
 connectable = engine_from_config(
 config.get_section(config.config_ini_section, {}),
 prefix="sqlalchemy.",
 poolclass=pool.NullPool,
)

 with connectable.connect() as connection:
 context.configure(
 connection=connection,
 target_metadata=target_metadata,
 render_as_batch=True, # Required for SQLite ALTER TABLE
)

 with context.begin_transaction():
 context.run_migrations()

if context.is_offline_mode():
 run_migrations_offline()
else:
 run_migrations_online()

```

5. Generate and apply the initial revision:

```

uv run alembic revision --autogenerate -m "create movies"
uv run alembic upgrade head
uv run alembic current # optional sanity check

```

## Step 4: Seed script

movie\_service/scripts/seed\_db.py

```

"""Seed the database with initial movie data."""

from sqlmodel import Session, select

from movie_service.app.database import engine, init_db
from movie_service.app.models import Movie, MovieCreate
from movie_service.app.repository_db import MovieRepository

def seed_movies() -> None:
 """Seed the database with sample movies if empty."""
 # Ensure tables exist
 init_db()

 with Session(engine) as session:
 # Check if database already has data
 statement = select(Movie)
 existing_movies = session.exec(statement).first()

 if existing_movies:

```

```

 print("Database already contains movies. Skipping seed.")
 return

Seed initial movies
repo = MovieRepository(session)
movies = [
 MovieCreate(title="Arrival", year=2016, genre="sci-fi"),
 MovieCreate(title="The Martian", year=2015, genre="sci-fi"),
 MovieCreate(title="Interstellar", year=2014, genre="sci-fi"),
]

for movie_data in movies:
 repo.create(movie_data)

print(f"Successfully seeded {len(movies)} movies.")

if __name__ == "__main__":
 seed_movies()

```

## Step 5: Verification checklist

- `uv run pytest movie_service/tests -v`
- `uv run alembic upgrade head`
- `uv run python -m movie_service.scripts.seed_db` (run twice to verify idempotency)
- Data persists in `data/movies.db` across uvicorn restarts
- README/docs updated with the new commands
- `.gitignore` contains `data/` and migration revision files are tracked (do not ignore `migrations/versions/`)

## Next steps

- Install Docker Desktop and verify `docker compose version` if you plan to run Postgres locally.
- Commit/tag this checkpoint (e.g., `session-04-complete`) for easy rollback.
- Capture current `.env` values so you can swap URLs cleanly when changing databases.
- Rerun `uv run pytest -v` to ensure a clean baseline before extending the stack.

## Facilitation Tips

- Open each target file before coding so everyone understands where the changes land.
- After Step 2, confirm `data/movies.db` exists; failing to do so causes confusing POST errors later.
- Encourage `uv run pytest movie_service/tests -k movies` mid-lab to keep failures tight.
- When introducing Alembic, show the generated revision file and connect it to the SQL it would run.
- Close by re-running the seed script + `/movies` `curl` so students see durable data.

## Troubleshooting

- `sqlite3.OperationalError: attempt to write a readonly database` → ensure `data/` is writable and ignored by Git.
- `check_same_thread` errors → confirm `connect_args={"check_same_thread": False}` is present anywhere FastAPI or tests create SQLite engines.

- **Alembic cannot locate metadata** → import `movie_service.app.models` in `env.py` and ensure `SQLModel.metadata` is referenced.
- **Tests mutate the real database** → verify the dependency override fixture runs before `TestClient` is instantiated.

## AI Prompt Seeds

---

- “Convert these Pydantic models + dict repository into SQLModel classes with a SQLite-backed repository; keep FastAPI route signatures identical.”
- “Write pytest fixtures that spin up a temporary SQLite database, override FastAPI dependencies, and keep each test hermetic.”
- “Draft an Alembic workflow (init, revision, upgrade) and a Typer/uv command list so students can recreate today’s migration + seed steps.”

# Session 05 – Multi-DB Modes + Postgres Hardening

---

docs/sessions/session-05.md

- **Date:** Monday, 30/03/2026
- **Theme:** Keep the FastAPI contract stable while making the repository facade run in three modes: in-memory, SQLite, and Postgres. Add health/CORS/trace protections and keep migrations/tests working regardless of backend.

## Session Story

---

Session 03 shipped the in-memory backend; Session 04 swapped in SQLite + Alembic without touching routes. Session 05 makes that abstraction explicit: you choose the datastore via config, keep the same models/routes, and prove the same migrations/tests run against SQLite (offline) and Postgres (compose). Memory stays available for ultra-fast unit tests. The outcome is a single codebase that can run everywhere—from ephemeral tests to classroom laptops to production-colored Postgres.

## Learning Objectives

---

- Expose a `db_mode` switch (memory | sqlite | postgres) with sensible defaults so environments pick their own backend without code edits.
- Drive SQLAlchemy/Alembic from environment URLs and keep the migration history identical across SQLite and Postgres.
- Maintain the repository facade so FastAPI routes never import storage-specific classes.
- Add health, CORS, and trace ID middleware that works in every mode.
- Seed and test against all backends with repeatable commands.

## Deliverables (What You'll Build)

---

- Env-driven settings (`MOVIE_DB_MODE`, `MOVIE_DATABASE_URL_SQLITE`, `MOVIE_DATABASE_URL_POSTGRES`, `MOVIE_DATABASE_URL_TEST`, pool flags) captured in `.env.example`, plus a computed `database_url` property.
- `docker-compose.yml` (provided in repo root) for Postgres with a persisted volume.
- `movie_service/app/database.py` that resolves the correct engine per mode (memory → no engine, sqlite → file path + `check_same_thread`, postgres → `psycopg` + pooling).
- `movie_service/app/dependencies.py` with a repository factory/protocol that returns the in-memory repository or the SQLAlchemy-backed repository without changing route signatures.
- Alembic configured to read URLs from settings; the same migration set applies to SQLite and Postgres.
- Typer/Rich CLI for seeding/resetting the chosen backend.
- Pytest fixtures that default to memory, can target SQLite files, and optionally spin up throwaway Postgres DBs.

## Toolkit Snapshot

---

- **FastAPI + SQLAlchemy** – shared models for API + persistence.
- **Alembic** – migration history that points at the active `MOVIE_DATABASE_URL`.
- **psycopg 3** – Postgres driver; only used when `db_mode=postgres`.
- **Docker Compose** – reproducible Postgres for local/dev demos.
- **pytest** – fixtures per mode; dependency overrides for repositories.

- **Typer/Rich** – CLI UX for seeds/resets regardless of backend.

## Before Class (JiTT)

0. Workflow reminder: keep briefs per slice under the 150-LOC limit in `docs/workflows/ai-assisted/README.md`.
1. Finish Session 04 and confirm SQLite path is green: `uv run pytest movie_service/tests -v`.
2. Add Postgres + tooling (needed only for postgres mode):

```
docker compose version
uv add "psycopg[binary]" rich typer
```

Add `sqlalchemy-utils` only if you rely on its Postgres helpers for throwaway DB creation.

3. Extend env templates:

```
MOVIE_DB_MODE="sqlite" # options: memory | sqlite | postgres
MOVIE_DATABASE_URL_SQLITE="sqlite:///./data/movies.db"
MOVIE_DATABASE_URL_POSTGRES="postgresql+psycopg://movie:movie@localhost:5432/movies"
MOVIE_DATABASE_URL_TEST="sqlite:///./data/movies-test.db" # override per test run
MOVIE_DATABASE_ECHO=false
MOVIE_POOL_SIZE=5
MOVIE_POOL_TIMEOUT=30
```

Keep `MOVIE_DATABASE_URL` as a computed property in code (see Lab 1). `.env.example` includes these keys; copy it to `.env` and adjust paths before class.

4. Create/confirm writable `data/` directory and ignore it in git.
5. (Optional) Start Postgres for the postgres path: Compose file lives at repo root ( `docker-compose.yml` ).

```
docker compose up -d db
pg_isready -h localhost -p 5432 -d movies -U movie
```

6. Smoke imports:

```
uv run python - <<'PY'
import sqlmodel, psycopg, typer # noqa: F401
print("SQLModel + psycopg + Typer ready")
PY
```

7. Baseline: `uv run pytest movie_service/tests -v` should still pass in default memory/sqlite mode before class.

## Session Agenda

Segment	Duration	Format	Focus
Recap & intent	10 min	Discussion	Why we keep three storage modes; how the facade prevents churn.
<b>Part A – Mode switch design</b>	20 min	Board + talk	Settings + dependency diagram for memory/sqlite/postgres.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>Wire config + dependencies for multi-DB, add health/CORS/trace.</b>
Break	10 min	—	Confirm URLs + logs per mode.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided testing</b>	<b>Migrations, seeds, pytest matrix (memory/sqlite/postgres).</b>
Wrap-up	10 min	Q&A	Checklist + next experiments.

## Lab 1 – Wire Multi-DB Modes (45 min)

Goal: make the repository + dependency layer choose memory, SQLite, or Postgres based on settings while keeping the HTTP contract unchanged.

Keep the interfaces identical: `RepositoryProtocol.list/create/get/delete` is the only shape routes see.

### Step 0 – (Optional) Boot Postgres

```
docker compose up -d db
pg_isready -h localhost -p 5432 -d movies -U movie
docker compose logs db | tail
```

`docker-compose.yml` ships with `postgres:15-alpine`, user/password `movie`, DB `movies`, and a persisted `pgdata` volume.

### Step 1 – Settings with a mode switch

`movie_service/app/config.py`

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
 app_name: str = "Movie Service"
 db_mode: str = "sqlite" # memory | sqlite | postgres
 database_url_sqlite: str = "sqlite:///./data/movies.db"
 database_url_postgres: str = "postgresql+psycopg://movie:movie@localhost:5432/movies"
 database_url_test: str | None = None
 database_echo: bool = False
 pool_size: int = 5
 pool_timeout: int = 30

 @property
 def database_url(self) -> str:
 if self.db_mode == "postgres":
 return self.database_url_postgres
 return self.database_url_sqlite

model_config = SettingsConfigDict(env_prefix="MOVIE_", env_file=".env", extra="ignore")
```

### Step 2 – Engine + session factory per mode

`movie_service/app/database.py`

```
from collections.abc import Generator
from contextlib import nullcontext
from typing import Annotated, Optional

from fastapi import Depends
from sqlmodel import Session, SQLModel, create_engine

from .config import Settings

def get_settings() -> Settings:
 return Settings()
```

```

SettingsDep = Annotated[Settings, Depends(get_settings)]

def _build_engine(settings: Settings):
 if settings.db_mode == "memory":
 return None
 url = settings.database_url_test or settings.database_url
 kwargs: dict = {"echo": settings.database_echo}
 if url.startswith("sqlite"):
 kwargs["connect_args"] = {"check_same_thread": False}
 if url.startswith("postgresql"):
 kwargs.update(
 pool_size=settings.pool_size,
 pool_timeout=settings.pool_timeout,
 pool_pre_ping=True,
)
 return create_engine(url, **kwargs)

engine = _build_engine(Settings())

def init_db() -> None:
 from . import models # noqa: F401

 if engine:
 SQLAlchemyModel.metadata.create_all(engine)

def get_session() -> Generator[Optional[Session], None, None]:
 if engine is None:
 with nullcontext():
 yield None # repository factory ignores session in memory mode
 return
 with Session(engine) as session:
 yield session

SessionDep = Annotated[Optional[Session], Depends(get_session)]

```

### Step 3 – Repository facade (memory + SQLAlchemyModel)

movie\_service/app/dependencies.py

```

from typing import Annotated, Protocol

from fastapi import Depends

from .database import SessionDep, SettingsDep
from .repository import MovieRepository as InMemoryRepository
from .repository_db import MovieRepository as SqlRepository

class MovieRepositoryProtocol(Protocol):
 def list(self, *, skip: int = 0, limit: int = 100): ...
 def create(self, payload): ...
 def get(self, movie_id: int): ...
 def delete(self, movie_id: int): ...

def get_repository(settings: SettingsDep, session: SessionDep) -> MovieRepositoryProtocol:
 if settings.db_mode == "memory":

```

```

 return InMemoryRepository()
 if session is None:
 raise RuntimeError("Database session required for non-memory modes")
 return SqlRepository(session)

```

```
RepositoryDep = Annotated[MovieRepositoryProtocol, Depends(get_repository)]
```

## Step 4 – Health/CORS/trace that works in every mode

movie\_service/app/main.py excerpt:

```

import uuid

from fastapi import FastAPI, HTTPException, Request, status
from fastapi.middleware.cors import CORSMiddleware
from sqlalchemy import text

from .database import engine, SettingsDep
from .dependencies import RepositoryDep
from .models import MovieCreate, MovieRead

app = FastAPI(title="Movie Service", version="0.5.0")
app.add_middleware(
 CORSMiddleware,
 allow_origins=["http://localhost:8501", "http://localhost:5173"],
 allow_credentials=True,
 allow_methods=["*"],
 allow_headers=["*"],
)

@app.middleware("http")
async def add_trace_id(request: Request, call_next):
 trace_id = request.headers.get("X-Trace-Id") or f"req-{uuid.uuid4().hex[:8]}"
 request.state.trace_id = trace_id
 response = await call_next(request)
 response.headers["X-Trace-Id"] = trace_id
 return response

@app.get("/healthz", tags=["health"])
def healthcheck(settings: SettingsDep) -> dict[str, str]:
 if engine:
 with engine.connect() as connection:
 connection.execute(text("SELECT 1"))
 backend = settings.db_mode if engine else "memory"
 return {"status": "ok", "database": backend}

```

Reuse the Session 03/04 CRUD routes unchanged; only the engine/middleware changed.

## Step 5 – Smoke test each mode

```

Memory (fastest for demos/tests)
MOVIE_DB_MODE=memory uv run uvicorn movie_service.app.main:app --reload

SQLite (offline durable)
MOVIE_DB_MODE=sqlite uv run uvicorn movie_service.app.main:app --reload

Postgres (production-colored)
MOVIE_DB_MODE=postgres uv run uvicorn movie_service.app.main:app --reload

```

```
curl -i -H "X-Trace-Id: smoke-1" http://localhost:8000/healthz
curl http://localhost:8000/movies
```

## Lab 2 – Migrations, Seeds, Tests per Mode (45 min)

Goal: run the same migration/seed/test story regardless of backend.

One migration history, multiple URLs. Never fork schemas per mode.

### Step 1 – Alembic points at settings

1. Ensure `alembic.ini` leaves `sqlalchemy.url` empty; `migrations/env.py` should read:

```
from sqlmodel import SQLModel

from movie_service.app.config import Settings

settings = Settings()
config.set_main_option("sqlalchemy.url", settings.database_url)
target_metadata = SQLModel.metadata
```

2. Commands per backend:

```
MOVIE_DB_MODE=sqlite uv run alembic upgrade head
MOVIE_DB_MODE=postgres uv run alembic upgrade head
```

### Step 2 – Typer seed/reset that respects the mode

`scripts/db.py` excerpt (new Typer CLI; replaces the Session 04 `seed_db` helper):

```
import typer
from sqlmodel import Session

from movie_service.app.config import Settings
from movie_service.app.database import engine, init_db
from movie_service.app.models import MovieCreate
from movie_service.app.repository import MovieRepository as MemoryRepo
from movie_service.app.repository_db import MovieRepository as DbRepo

app = typer.Typer(help="Database utilities")

@app.command()
def bootstrap(sample: int = 5) -> None:
 settings = Settings()
 if settings.db_mode == "memory":
 repo = MemoryRepo()
 for idx in range(sample):
 repo.create(MovieCreate(title=f"Sample {idx+1}", year=2010 + idx, genre="sci-fi"))
 typer.echo("Seeded in-memory repo (non-persistent).")
 return

 init_db()
 with Session(engine) as session:
 repo = DbRepo(session)
 if repo.list():
 typer.echo("Database already seeded; skipping.")
 return
 for idx in range(sample):
```

```

repo.create(MovieCreate(title=f"Sample {idx+1}", year=2010 + idx, genre="sci-fi"))
session.commit()
typer.echo(f"Seeded {sample} movies in {settings.db_mode}.")

if __name__ == "__main__":
 app()

```

Place this file at `scripts/db.py` in your project root (or under `movie_service/scripts/db.py` if you keep scripts inside the package); adjust imports to match your layout. Run: `MOVIE_DB_MODE=sqlite uv run python scripts/db.py bootstrap --sample 3` (swap to postgres as needed).

### Step 3 – pytest matrix

- **Memory (default):**

```
MOVIE_DB_MODE=memory uv run pytest movie_service/tests -v
```

- **SQLite (temp file per test module):** reuse Session 04 fixtures; ensure `database_url_test` points to a temp path and drop the file in teardown.
- **Postgres (optional):** reuse the throwaway DB fixture from below; ensure the `movie` role has `CREATEDB`.

`movie_service/tests/conftest.py` excerpt for Postgres:

```

import uuid

import psycopg
import pytest
from sqlmodel import SQLModel, Session, create_engine

from movie_service.app.dependencies import get_repository
from movie_service.app.main import app
from movie_service.app.models import Movie
from movie_service.app.repository_db import MovieRepository

ADMIN_URL = "postgresql://movie:movie@localhost:5432/postgres"
DB_TEMPLATE = "postgresql+psycopg://movie:movie@localhost:5432/{db_name}"

```

(Use the same create/drop helpers from Session 04; swap URL based on `MOVIE_DB_MODE`.)

### Step 4 – Regression commands

```

MOVIE_DB_MODE=sqlite uv run alembic upgrade head
MOVIE_DB_MODE=sqlite uv run python scripts/db.py bootstrap --sample 3
MOVIE_DB_MODE=memory uv run pytest movie_service/tests -v
MOVIE_DB_MODE=postgres uv run pytest movie_service/tests -v # optional if Postgres running

```

## Wrap-Up & Success Criteria

- ☐ `MOVIE_DB_MODE` toggles between memory/sqlite/postgres without code changes.
- ☐ `/healthz` reports the active backend and echoes `X-Trace-Id`.
- ☐ Alembic `upgrade head` succeeds for SQLite and Postgres URLs.
- ☐ Seed script is idempotent per backend.
- ☐ Pytest passes in memory mode; optional runs green for SQLite/Postgres fixtures.

- README/docs updated with Compose/Alembic/seed/test commands per mode.

## Next steps

---

- Keep `docker compose ps` handy to ensure Postgres is healthy when using that mode.
- Add CI matrix jobs for memory + SQLite; gate Postgres behind an opt-in service job.
- Wire Streamlit/React clients to the CORS/trace-aware API once health checks pass.

## Troubleshooting

---

- **psycopg.OperationalError: connection refused** → ensure Docker Desktop/Colima is running and port 5432 is free; restart with `docker compose down && docker compose up -d`.
- **permission denied to create database** → grant `CREATEDB` to the `movie` role: `docker compose exec db psql -c "ALTER ROLE movie CREATEDB;"`.
- **Alembic still targets SQLite** → delete hard-coded URLs in `alembic.ini` and verify `Settings().database_url` points to the chosen backend.
- **Tests hang on drop** → terminate active connections before `DROP DATABASE` (see `_drop_test_db`).
- **CORS blocked** → ensure `allow_origins` includes `http://localhost:8501` and `http://localhost:5173`.

# Session 06 – Streamlit Dashboards + Typer UX

---

docs/sessions/session-06.md

- **Date:** Monday, 13/04/2026
- **Theme:** Ship a Streamlit dashboard and Typer CLI on top of the Postgres-backed FastAPI API, plus a short JavaScript/TypeScript warm-up to prep for frontend work.

## Session Story

---

With Postgres live from Session 05, the backend is stable enough for real UIs. Today you'll build a Streamlit dashboard that lists, filters, and creates movies through the existing API, and you'll add Typer/Rich helpers for operators. We end with a short JavaScript/TypeScript warm-up so next week's React sprint feels familiar.

## Learning Objectives

---

- Consume FastAPI with typed `httpx` helpers, shared env vars, and trace IDs.
- Build Streamlit dashboards that cache GETs, bust cache on writes, and render quick insights.
- Automate seed/reset/export flows with a Typer CLI layered on the repository.
- Keep tests green against Postgres while UI layers evolve.
- Refresh core JS/TS concepts (modules, async/await, fetch) to stay ready for frontend work.

## Deliverables (What You'll Build)

---

- `frontend/client.py` – typed HTTP client with trace headers for `/movies`.
- `frontend/dashboard.py` – Streamlit page with list view, filters, metrics, and create/delete flows.
- `scripts/ui.py` – Typer CLI for seed/reset/export (uses the same repository).
- Updated docs/runbook notes for running FastAPI + Streamlit concurrently.

## Toolkit Snapshot

---

- **Streamlit** – Python-first dashboards + forms.
- **httpx** – shared client for Streamlit and Typer.
- **Typer + Rich** – operator UX with nice output.
- **PostgreSQL** – same DB from Session 05; UI never talks to it directly.
- **JavaScript/TypeScript primer** – light prep for Vite/React.

## Before Class (JiTT)

---

1. Ensure Postgres is running: `docker compose up -d db` and `curl http://localhost:8000/healthz`.
2. `.env` should include `MOVIE_API_BASE_URL=http://localhost:8000` plus the Postgres URL from Session 05.
3. Install UI deps:

```
uv add streamlit rich typer httpx python-dotenv pandas
```

4. Seed some movies (reuse `scripts/db.py`): `uv run python scripts/db.py bootstrap --sample 5`.

5. JS preview sanity checks (for the short demo): `node --version` , `corepack enable` , `corepack prepare pnpm@latest --activate` , `pnpm --version` .

## Agenda

Segment	Duration	Format	Focus
Recap & intent	10 min	Discussion	Why Postgres unlocks UI experiments.
Streamlit + API primer	20 min	Talk + demo	Typed clients, caching, trace IDs, CORS.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>Client helpers + dashboard list view.</b>
Break	10 min	—	Cache/CORS Q&A.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided build</b>	<b>Forms, Typer CLI, runbook updates.</b>
JS/TS preview	15 min	Talk + mini-demo	Async/await, fetch, pnpm basics.
Wrap-up	10 min	Q&A	Checklist and open questions.

## Lab 1 – Typed client + dashboard (45 min)

Goal: render Postgres-backed movies in Streamlit with cached GETs and ready-to-wire write paths.

### Step 0 – Confirm API + data

Keep FastAPI running in one terminal:

```
uv run uvicorn movie_service.app.main:app --reload
curl http://localhost:8000/movies
```

Seed if empty: `uv run python scripts/db.py bootstrap --sample 5` .

### Step 1 – Typed client

frontend/client.py

```
from __future__ import annotations

from functools import lru_cache
from typing import Any

import httpx
from pydantic_settings import BaseSettings, SettingsConfigDict

class UISettings(BaseSettings):
 api_base_url: str = "http://localhost:8000"
 trace_id: str = "ui-streamlit"

 model_config = SettingsConfigDict(env_prefix="MOVIE_", env_file=".env", extra="ignore")

settings = UISettings()

@lru_cache(maxsize=1)
```

```

def _client() -> httpx.Client:
 return httpx.Client(
 base_url=settings.api_base_url,
 headers={"X-Trace-Id": settings.trace_id},
 timeout=5.0,
)

def list_movies(*, genre: str | None = None) -> list[dict[str, Any]]:
 params = {"genre": genre} if genre else None
 response = _client().get("/movies", params=params)
 response.raise_for_status()
 return response.json()

def create_movie(*, title: str, year: int, genre: str) -> dict[str, Any]:
 response = _client().post("/movies", json={"title": title, "year": year, "genre": genre})
 response.raise_for_status()
 return response.json()

def delete_movie(movie_id: int) -> None:
 response = _client().delete(f"/movies/{movie_id}")
 response.raise_for_status()

```

## Step 2 – First Streamlit view

frontend/dashboard.py

```

import pandas as pd
import streamlit as st

from frontend.client import list_movies

st.set_page_config(page_title="Movie Pulse", layout="wide")
st.title("Movie Service Dashboard")
st.caption("FastAPI + Postgres backing; trace IDs flow through headers.")

@st.cache_data(ttl=30)
def cached_movies(genre: str | None = None) -> list[dict]:
 return list_movies(genre=genre)

with st.spinner("Fetching movies..."):
 movies = cached_movies()

if not movies:
 st.info("No movies yet. Add some via the API or Typer CLI.")
else:
 st.metric("Total movies", len(movies))
 st.dataframe(pd.DataFrame(movies))

```

Run: `uv run streamlit run frontend/dashboard.py`.

## Step 3 – Filters + cache invalidation

```

genres = sorted({m["genre"] for m in movies})
selected = st.multiselect("Filter by genre", genres)
displayed = cached_movies(selected[0] if selected else None)

```

```

if st.button("Refresh data"):
 cached_movies.clear()
 st.experimental_rerun()

st.dataframe(pd.DataFrame(displayed))

```

## Lab 2 – Forms, Typer, documentation (45 min)

Goal: mutate data from the UI, add an operator CLI, and capture how to run both servers.

### Step 1 – Write operations

In `frontend/dashboard.py`, add:

```

from frontend.client import create_movie, delete_movie

with st.expander("Add movie"):
 with st.form("create_movie"):
 title = st.text_input("Title")
 year = st.number_input("Year", min_value=1900, max_value=2100, value=2024)
 genre = st.text_input("Genre", value="sci-fi")
 submitted = st.form_submit_button("Create")
 if submitted:
 try:
 movie = create_movie(title=title, year=year, genre=genre)
 cached_movies.clear()
 st.success(f"Created {movie['title']}")
 except Exception as exc: # pragma: no cover
 st.error(f"Failed: {exc}")

if movies:
 to_delete = st.selectbox("Delete movie", options=[m["id"] for m in movies])
 if st.button("Delete selected"):
 delete_movie(int(to_delete))
 cached_movies.clear()
 st.success("Deleted movie")

```

### Step 2 – Typer CLI for operators

`scripts/ui.py` (describe; create during lab):

```

import typer
from rich import print

from movie_service.app.models import MovieCreate
from movie_service.app.repository_db import MovieRepository
from movie_service.app.database import engine, init_db
from sqlmodel import Session

app = typer.Typer(help="UI/ops helpers")

@app.command()
def reset(sample: int = 5) -> None:
 """Drop + recreate tables; seed sample movies."""
 init_db()
 with Session(engine) as session:
 repo = MovieRepository(session)
 repo.delete_all()

```

```

 for idx in range(sample):
 repo.create(MovieCreate(title=f"Sample {idx+1}", year=2010 + idx, genre="sci-fi"))
 print(f"[green]Reset complete with {sample} movies[/green]")

@app.command()
def export() -> None:
 with Session(engine) as session:
 repo = MovieRepository(session)
 for movie in repo.list():
 print(movie)

if __name__ == "__main__":
 app()

```

Run: `uv run python scripts/ui.py reset --sample 5`.

### Step 3 – JS/TS preview (15 minutes)

- Verify Node/pnpm ( `node --version` , `pnpm --version` ).
- Mini fetch demo (run with `node` ):

```

const res = await fetch("http://localhost:8000/movies");
const movies = await res.json();
console.log(movies.map((m) => m.title));

```

- Emphasize that any future frontend work will reuse the same API and trace headers.

## Wrap-Up & Success Criteria

- ☐ FastAPI + Postgres healthy; `/healthz` returns OK with trace header.
- ☐ Streamlit dashboard lists movies, filters, and creates/deletes entries (cache clears on writes).
- ☐ Typer CLI can reset/seed/export without touching production data.
- ☐ README/runbook notes mention how to run FastAPI and Streamlit side-by-side.
- ☐ JS/TS preview commands verified for the mini-demo.

## Next steps

- Keep Postgres + `/healthz` handy.
- Install Node 20 + pnpm if you plan to build a React/Vite variant.
- Note the existing API contract ( `/movies` list/create/get/delete) for any frontend you add next.

## Troubleshooting

- **Streamlit cannot fetch** → verify FastAPI is running and CORS allows `http://localhost:8501` .
- **Cache not updating** → call `cached_movies.clear()` after POST/DELETE and rerun the script.
- **Typer commands fail on missing tables** → run `uv run alembic upgrade head` or `init_db()` first.
- **pnpm errors** → re-run `corepack enable` and ensure Node  $\geq 20$ .

# Session 07 – React/Vite Foundations + Reliability Upgrades

docs/sessions/session-07.md

- **Date:** Monday, 27/04/2026
- **Theme:** Build a Vite/React (TypeScript) client that reuses the FastAPI/Postgres API, then tighten reliability with frontend checks and stronger backend tests/observability.

## Session Story

Session 06 delivered Streamlit + Typer. Session 07 finishes the UI runway with React + Vite while keeping the same `/movies` contract. Once the React client works, we add lightweight lint/tests for the frontend and expand backend reliability (pytest improvements, trace logging) so multiple clients stay stable.

## Learning Objectives

- Scaffold a Vite + React TypeScript app with pnpm and consume the existing API.
- Centralize API calls in a service layer with trace headers.
- Manage data fetching/mutation with React Query and simple forms.
- Add frontend lint/tests (ESLint, Vitest/Testing Library) to catch regressions.
- Strengthen backend reliability with parametrized tests and trace-aware logging.

## Deliverables (What You'll Build)

- `frontend-react/` Vite project (React + TS) using the `/movies` API.
- `src/lib/api.ts`, `src/services/movies.ts`, `src/hooks/useMovies.ts` service layer + hook.
- `src/App.tsx` list/create UI with React Query invalidation.
- Frontend scripts for `dev`, `lint`, and `test`.
- Backend reliability add-ons: parametrized tests + trace-aware logging (reuse Postgres fixtures).

## Toolkit Snapshot

- **Node 20 + pnpm** – runtime/package manager for React.
- **Vite + React + TS** – frontend build + framework.
- **Axios or fetch + React Query** – API client + caching.
- **ESLint + Vitest + Testing Library** – frontend quality checks.
- **pytest + existing Postgres fixtures** – backend regression safety.

## Before Class (JiTT)

1. Postgres + FastAPI running ( `docker compose up -d db`, `curl http://localhost:8000/healthz` ).
2. Node ≥ 20 with pnpm ready:

```
node --version
corepack enable
```

```
corepack prepare pnpm@latest --activate
pnpm --version
```

3. Install backend reliability extras (if missing):

```
uv add pytest-cov hypothesis logfire
```

4. Keep `.env` with `MOVIE_API_BASE_URL` for parity across Streamlit/React.

## Agenda

Segment	Duration	Format	Focus
JS/TS warm-up	15 min	Talk + mini-demo	Modules, async/await, fetch vs axios.
Vite setup	15 min	Walkthrough	Scaffold project, env vars, scripts.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>Service layer + React Query UI.</b>
Break	10 min	—	Fix pnpm/env issues.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided reliability</b>	<b>Frontend lint/tests + backend test boosts.</b>
Wrap-up	10 min	Discussion	Checklist and discussion.

## Lab 1 – Scaffold Vite + movie client (45 min)

Goal: run `pnpm dev`, see Postgres-backed movies, and create new entries using the same trace headers Streamlit used.

### Step 0 – Create project

```
cd hello-uv
pnpm create vite@latest frontend-react -- --template react-ts
cd frontend-react
pnpm install
```

Add `.env.local`:

```
VITE_API_BASE_URL=http://localhost:8000
VITE_TRACE_ID=ui-react
```

### Step 1 – Install UI deps

```
pnpm add axios @tanstack/react-query
pnpm add -D eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin vitest @testing-
library/react @testing-library/jest-dom @testing-library/user-event
```

### Step 2 – API client + services

```
src/lib/api.ts
```

```
import axios from "axios";
```

```

export const client = axios.create({
 baseURL: import.meta.env.VITE_API_BASE_URL,
 headers: { "X-Trace-Id": import.meta.env.VITE_TRACE_ID ?? "ui-react" },
 timeout: 5000,
});

export async function get<T>(url: string, params?: Record<string, unknown>) {
 const response = await client.get<T>(url, { params });
 return response.data;
}

export async function post<T>(url: string, body: unknown) {
 const response = await client.post<T>(url, body);
 return response.data;
}

```

src/services/movies.ts

```

import { get, post } from "../lib/api";

export type Movie = { id: number; title: string; year: number; genre: string };

export function listMovies(genre?: string) {
 return get<Movie[]>("/movies", genre ? { genre } : undefined);
}

export function createMovie(payload: Omit<Movie, "id">) {
 return post<Movie>("/movies", payload);
}

```

src/hooks/useMovies.ts

```

import { useMutation, useQuery, useQueryClient } from "@tanstack/react-query";
import { createMovie, listMovies, Movie } from "../services/movies";

export function useMovies(genre?: string) {
 const qc = useQueryClient();

 const listQuery = useQuery({
 queryKey: ["movies", genre],
 queryFn: () => listMovies(genre),
 });

 const create = useMutation({
 mutationFn: (payload: Omit<Movie, "id">) => createMovie(payload),
 onSuccess: () => qc.invalidateQueries({ queryKey: ["movies"] }),
 });

 return { ...listQuery, create };
}

```

Wrap React Query in `src/main.tsx` :

```

import React from "react";
import ReactDOM from "react-dom/client";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import App from "./App";

const qc = new QueryClient();

ReactDOM.createRoot(document.getElementById("root")!).render(

```

```

<React.StrictMode>
 <QueryClientProvider client={qc}>
 <App />
 </QueryClientProvider>
</React.StrictMode>,
);

```

### Step 3 – Replace `src/App.tsx`

```

import { FormEvent, useState } from "react";
import { useMovies } from "../hooks/useMovies";
import "../App.css";

export default function App() {
 const [genre, setGenre] = useState<string | undefined>();
 const { data, isLoading, isError, create } = useMovies(genre);

 const handleSubmit = (event: FormEvent<HTMLFormElement>) => {
 event.preventDefault();
 const form = new FormData(event.currentTarget);
 create.mutate({
 title: String(form.get("title")),
 year: Number(form.get("year")),
 genre: String(form.get("genre")),
 });
 event.currentTarget.reset();
 };

 if (isLoading) return <p>Loading movies...</p>;
 if (isError) return <p role="alert">Failed to load movies.</p>;

 return (
 <main>
 <header>
 <h1>Movie Pulse</h1>
 <p>Total movies: {data?.length ?? 0}</p>
 <label>
 Genre filter
 <input value={genre ?? ""} onChange={(e) => setGenre(e.target.value || undefined)} />
 </label>
 </header>

 <section>
 <form onSubmit={handleSubmit}>
 <input name="title" placeholder="Title" required />
 <input name="year" type="number" min={1900} max={2100} required />
 <input name="genre" placeholder="Genre" defaultValue="sci-fi" />
 <button type="submit" disabled={create.isLoading}>
 {create.isLoading ? "Saving..." : "Add movie"}
 </button>
 </form>
 </section>

 {data?.map((movie) => (
 <li key={movie.id}>
 {movie.title} ({movie.year}) - {movie.genre}

))}

 </main>
);
}

```

Run `pnpm dev`, visit `http://localhost:5173`, and verify movies list/create work against FastAPI/Postgres.

## Lab 2 – Reliability + tests (45 min)

Goal: keep quality high now that two UIs exist.

### Frontend lint/tests

- Add scripts to `package.json` :

```
"scripts": {
 "dev": "vite",
 "build": "tsc && vite build",
 "lint": "eslint src --ext .ts,.tsx",
 "test": "vitest"
}
```

- Sample test `src/App.test.tsx` :

```
import { afterEach, expect, it, vi } from "vitest";
import { render, screen, waitFor } from "@testing-library/react";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import App from "../App";
import * as svc from "../services/movies";

const qc = new QueryClient();
afterEach(() => qc.clear());

it("renders movies", async () => {
 vi.spyOn(svc, "listMovies").mockResolvedValue([
 { id: 1, title: "Arrival", year: 2016, genre: "Sci-Fi" },
]);
 render(
 <QueryClientProvider client={qc}>
 <App />
 </QueryClientProvider>,
);
 await waitFor(() => expect(screen.getByText(/Arrival/)).toBeInTheDocument());
});
```

### Backend reliability boosts

- Reuse Postgres fixtures from Session 05; add parametrized tests (year bounds, genre casing) in `movie_service/tests/test_movies.py`.
- Optional property tests with Hypothesis (ASCII titles) to catch edge cases.
- Ensure `X-Trace-Id` flows through logs; add basic structured logging if missing.

### Regression commands

```
pnpm lint && pnpm test
uv run pytest movie_service/tests -v
```

## Wrap-Up & Success Criteria

- ☐ React app lists and creates movies against FastAPI/Postgres with trace IDs.

- `pnpm lint` and `pnpm test` pass.
- Backend pytest suite still green after UI changes.
- Docs note how to run Streamlit or React with the same API.

## Next steps

---

- Keep `/healthz` green and Postgres running.
- Ensure React + Streamlit can both hit the API (CORS/trace).
- Install `uv add pydantic-ai httpx` (plus `dspy-ai` if you plan to explore agentic tools).

## Troubleshooting

---

- `pnpm dev` cannot reach API → verify FastAPI on port 8000 and CORS allows `http://localhost:5173`.
- **TypeScript references** `process.env` → use `import.meta.env.VITE_*` (Vite convention).
- **pytest fixtures drop DB slowly** → terminate active connections before `DROP DATABASE` (Session 05 helpers handle this).
- **Trace IDs missing** → ensure React client sends `X-Trace-Id` header (defaulted in `api.ts`).

# Session 08 – AI Sidecar for Dynamic Compute (Calculator + Codegen + SQL)

docs/sessions/session-08.md

- **Date:** Monday, 04/05/2026
- **Theme:** Stand up a simple FastAPI calculator backend (add/subtract + tiny charts), pair it with a Streamlit UI, and bolt on an AI sidecar (Pydantic AI + local vLLM or remote AI Studio) that generates and validates Python snippets for advanced calculations on demand—under guardrails—and a free-language-to-SQL agent that runs against a DuckDB microservice loaded from uploaded CSVs.

## Session Story

We pivot to a contained “calculator + AI” slice: start with a FastAPI calculator backend (basic math endpoints plus a tiny plotting endpoint), add a Streamlit UI that uploads CSVs and calls the calculator, then attach an AI sidecar (FastAPI + Pydantic AI) that can generate/validate Python code for advanced calculations via vLLM or Google AI Studio. We also load the uploaded CSV into a tiny DuckDB microservice so an SQL agent can translate free language into SQL over that table. The same sidecar hosts both TEXT→PY and TEXT→SQL flows. The stack stays Postgres-free for this session—just FastAPI, Streamlit, the sidecar, and DuckDB in-memory—so all AI hops stay simple.

## Architecture Snapshot

- **Frontend:** Streamlit (CSV upload, charts, free-text prompts for Python and SQL paths). **Talks only to the calculator backend.**
- **Calculator backend (FastAPI):** basic math + chart endpoints; advanced endpoints call the AI sidecar and DuckDB service; adds execution/telemetry envelopes. Acts as the sole gateway for the UI.
- **AI sidecar (FastAPI + Pydantic AI):** Python codegen + validator + refine loop; SQL codegen + validator + refine loop; exposes `/ai/codegen`, `/ai/sql`, `/ai/execute` (sandboxed run). Same service covers TEXT→PY and TEXT→SQL.
- **DuckDB microservice:** holds the uploaded CSV as `uploaded` table; exposes `/upload_csv` and `/query` (SELECT-only).
- **LLM providers:** local vLLM/LM Studio (optional container in compose) or remote Google AI Studio behind the sidecar via the same typed client.

## Build Plan (stepwise)

1. **Baseline backend + UI (no AI):** FastAPI `/healthz`, `/add`, `/subtract`, `/chart`; Streamlit calls only these; return `X-Trace-Id`.
2. **DuckDB smoke test:** Stand up DuckDB microservice; backend adds `/data/ping` (hard-coded SELECT 1) and `/data/query/raw` (fixed SQL) to verify connectivity; UI shows the ping result.
3. **AI sidecar (local LLM):** Add sidecar with `/ai/codegen` + `/ai/validate/python`; backend adds `/calculate/advanced` that calls the sidecar and returns the code + validation (no execution yet); UI shows generated code.
4. **CSV → DuckDB:** Backend exposes `/data/upload` to forward CSV to DuckDB microservice; UI uploads through backend; store returned columns.
5. **Free-language → SQL:** Sidecar gets `/ai/sql` + validator; backend `/calculate/sql` calls sidecar then DuckDB; UI sends free text, receives SQL + rows.

6. **Code execution + visualization:** Add `/ai/execute` sandbox; backend executes validated code; UI can run text-to-Python for analysis/plots; show suggested prompts.
7. **Refinement + telemetry:** Enable refine loops, rate limits, observability; lock down safety (SQL + Python).

## Docker/Dev Layout (outline only)

---

- Use `docker-compose.yml` with four services:
  - `calculator` : FastAPI main backend ( `uvicorn app.main:app` ), depends on `duckdb` and `sidecar` .
  - `sidecar` : FastAPI AI sidecar ( `uvicorn sidecar.main:app` ), environment for LLM endpoint ( `AI_BASE_URL` , `AI_API_KEY` , `AI_MODEL` ).
  - `duckdb` : DuckDB microservice ( `uvicorn duckdb_service.main:app` ), mounts a writable volume for temp CSVs if needed.
  - `streamlit` : Frontend ( `streamlit run ui/app.py` ), depends on `calculator` .
- Optional: `llm-local` : LM Studio/vLLM-compatible API on `/v1` for local track; set `AI_BASE_URL=http://llm-local:8000/v1` when enabled.
- Shared network: `app-net` ; expose ports 8000 (calculator), 8001 (duckdb), 8002 (sidecar), 8501 (streamlit).
- Volumes: optional `duckdb_data` for persisted duckdb files; otherwise in-memory.
- Env files: `.env` loaded into `calculator` and `sidecar` for AI keys/model/base URL; `DUCKDB_SERVICE_URL=http://duckdb:8001` ; include `GOOGLE_API_KEY` , `GOOGLE_GEMINI_MODEL` for remote track.
- Healthchecks: `curl -f http://localhost:PORT/healthz` for calculator/sidecar/duckdb; Streamlit basic HTTP check.
- Dev loop: `docker compose up --build` ; tests run inside `calculator` container with `pytest` .

## Learning Objectives

---

- Build a minimal FastAPI calculator service (add/subtract and a small chart endpoint) and keep routes stable.
- Stand up an AI sidecar (FastAPI + Pydantic AI) that generates Python snippets for advanced calculations and exposes them over HTTP to the calculator backend.
- Use validator + refinement agents to lint/approve and iteratively improve generated Python (reduce hallucinations, enforce safe imports/patterns) before execution.
- Translate free language to SQL against a DuckDB microservice loaded from CSV uploads; validate SQL and map results back to the UI.
- Wire Streamlit to upload CSVs, call calculator endpoints, and request sidecar codegen/SQL for custom analysis.
- Run both local (vLLM) and remote (Google AI Studio) LLMs through the same typed interface; cover with pytest/mocked transports and telemetry.

## Deliverables (What You'll Build)

---

- Prompt templates/checklists for spec-first and tests-first requests.
- A FastAPI calculator service (basic math + tiny chart endpoint) with tests.
- An AI sidecar FastAPI service exposing Pydantic AI tools for code generation, validation, and SQL translation.
- DuckDB microservice that ingests CSV uploads and exposes a simple `/query` endpoint used by the SQL agent.
- Streamlit UI that uploads CSVs, hits calculator endpoints, and calls the sidecar for custom analysis (Python or SQL).
- pytest cases that mock LLM responses and verify validator gating for both Python and SQL; optional DSPy scratch file.

## Toolkit Snapshot

- **FastAPI** – calculator backend (core) + AI sidecar (codegen/validation gateway).
- **Pydantic AI** – typed tools/agents for code generation, SQL translation, and validation/refinement.
- **DuckDB + tiny HTTP microservice** – holds the uploaded CSV as an in-memory table for SQL agent execution.
- **vLLM / Google AI Studio** – interchangeable LLM endpoints behind the sidecar.
- **httpx** – shared HTTP client between services.
- **Streamlit** – CSV upload + calculator UI + AI-assisted analysis.
- **pytest** – mocks LLM transports and tests validator gating.
- **Logfire/structured logging** – telemetry with `X-Trace-Id` across backend + sidecar + DuckDB microservice.

## Before Class (JiTT)

1. FastAPI calculator + DuckDB microservice running; `/healthz` should echo `X-Trace-Id`. No Postgres needed in this session.
2. Install deps:

```
uv add fastapi uvicorn pydantic-ai httpx duckdb pandas numpy streamlit plotly
Optional DSPy:
uv add dspy-ai
```

3. DuckDB microservice setup:

```
duckdb_service/main.py
from fastapi import FastAPI, UploadFile, HTTPException
from pydantic import BaseModel
import duckdb
import pandas as pd

app = FastAPI()
conn = duckdb.connect(":memory:")

class QueryRequest(BaseModel):
 sql: str
 params: dict[str, str] = {}

@app.post("/upload_csv")
async def upload_csv(file: UploadFile):
 df = pd.read_csv(file.file)
 conn.register("uploaded", df)
 return {"columns": df.columns.tolist(), "rows": len(df)}

@app.post("/query")
async def query(req: QueryRequest):
 # Enforce SELECT-only
 if not req.sql.strip().upper().startswith("SELECT"):
 raise HTTPException(400, "Only SELECT queries allowed")
 result = conn.execute(req.sql, req.params).fetchdf()
 return {"rows": result.to_dict(orient="records"), "count": len(result)}
```

4. Pick one LLM track:
  - **Local vLLM**: run model, expose `/v1`, set `AI_BASE_URL=http://localhost:8000/v1`, `AI_API_KEY=dummy`.
  - **Google AI Studio (Gemini)**: set `GOOGLE_API_KEY`, `GOOGLE_GEMINI_MODEL`, install `uv add "pydantic-ai[google]" google-genai`.
5. Add AI env entries to `.env.example` / `.env` for the sidecar:

```

AI_BASE_URL="http://localhost:8000/v1"
AI_MODEL="local-model-or-gemini"
AI_API_KEY="your-key-or-dummy"
DUCKDB_SERVICE_URL="http://localhost:8001"
CODE_EXECUTION_TIMEOUT=10
CODE_EXECUTION_MAX_MEMORY_MB=256
For Google AI Studio track
GOOGLE_API_KEY="your-google-api-key"
GOOGLE_GEMINI_MODEL="gemini-1.5-flash"

```

#### 6. Stand up skeleton services before class:

- Calculator FastAPI service with `/add` , `/subtract` , `/chart` (matplotlib/plotly light) and tests.
- AI sidecar FastAPI service with `/ai/codegen` , `/ai/validate` , and `/ai/sql` hitting Pydantic AI tools (mock transport OK).
- DuckDB microservice reachable from calculator (or mounted in-process) with `/query` .
- Streamlit page loads and can call calculator and AI endpoints locally.

## Agenda

Segment	Duration	Format	Focus
Intent & safety	10 min	Discussion	Why AI is sidecarred; guardrails, attribution.
Calculator core	20 min	Live coding	FastAPI add/subtract + tiny chart endpoint + tests.
Streamlit upload	20 min	Live demo	CSV upload, call calculator API, show chart, load CSV into DuckDB.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>AI sidecar: Pydantic AI code + SQL tools, validator/refiner + <code>/ai/codegen</code> + <code>/ai/sql</code> routes.</b>
Break	10 min	—	Timer + Q&A.
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided build</b>	<b>Wire Streamlit to sidecar + DuckDB; run local vLLM vs Google AI Studio; pytest with mocked LLM.</b>
Wrap-up	10 min	Discussion	Telemetry, next steps, homework.

## Guardrails & Prompt Patterns

1. **Policy reminder:** No secrets or private data; document AI assistance in changelog/PR. Every change must be understood and tested by a human.
2. **Spec-first prompt:** "Given this API contract and schema, draft the implementation." Paste types/acceptance criteria.
3. **Tests-first prompt:** Ask for pytest cases before code; run them locally; only then request implementation.
4. **Refactor prompt:** "Keep behavior, improve structure," paired with current code + tests.
5. **Telemetry toggle:** Keep `LOGFIRE_API_KEY` optional; default to local structured logs with `X-Trace-Id` .
6. **SQL safety:** Validator agent enforces SELECT-only SQL; blocks DDL/DML keywords; restricts to `uploaded` table; requires explicit column names from schema; adds `LIMIT 200` if missing.
7. **Python safety:** Validator agent blocks unsafe imports (os, subprocess, socket, requests); forbids `eval/exec/compile`; enforces numpy/pandas/math only; checks for file I/O patterns; uses `ast.parse()` for syntax validation.

8. **Execution safety:** Code runs in sandboxed environment with resource limits (see Security Hardening section).
9. **Refine loop:** If validator flags Python or SQL, ask the LLM to propose a patch diff; re-run validator; cap retries at 2; log all attempts with trace ID.

## Prompt Pack (copy/paste)

---

- **Spec-first (backend route):** “Given this contract for `POST /calculate/advanced` that returns `CodegenResponse` and `ValidationResult`, draft FastAPI code using dependency-injected httpx clients and Pydantic models. Include X-Trace-Id propagation and 422 on validation failure.”
- **Tests-first (sidecar agents):** “Write pytest with httpx.MockTransport to cover `/ai/codegen` success, invalid JSON, and unsafe import rejection. Use fixtures for mock LLM responses.”
- **SQL safety check:** “Review this SQL for safety. Enforce SELECT-only, table `uploaded`, allowed columns: {columns}, must include LIMIT <= 1000. Respond with JSON: `is_valid (bool), issues (list)`.”
- **Python refine:** “Given validation issues {issues}, propose a minimal patch to this code to pass safety checks while keeping the behavior: {code}. Return only the patched code in fenced `python`.”

## Error Handling Matrix

---

Standardize HTTP status codes across services:

- **400:** Malformed request (invalid JSON, missing required fields)
- **422:** Validation failure (unsafe code, invalid SQL, schema violation)
- **500:** Execution error (code crash, DuckDB query failure)
- **503:** LLM timeout or unavailable
- **429:** Rate limit exceeded (if implemented)

All errors include `{"error": "message", "details": {...}, "trace_id": "..."}.`

## Lab 1 – Build the AI sidecar (45 min)

---

Goal: expose a safe codegen + validator sidecar (Python + SQL) that the calculator backend can call; cover it with pytest.

### Step 1 – Define settings + models (description)

- AI settings: `ai_base_url`, `ai_model`, `ai_api_key`, `ai_trace_id`, `duckdb_service_url`, `code_execution_timeout`, `code_execution_max_memory_mb`.
- Calculator models: `OperationRequest` (`op`, `a`, `b`), `ChartRequest` (list of numbers), `ChartResponse` (image/data URL).
- Sidecar models: `CodegenRequest` (operation description or CSV columns), `CodegenResponse` (`python_code`, `summary`, `allowed_imports`); `ValidationResult` (`is_valid`, `issues`, `severity`).
- SQL models: `SqlQueryRequest` (`prompt`, `available_columns`, `table_name="uploaded"`), `SqlQueryResponse` (`sql`, `summary`, `parameters`, `estimated_rows`).
- Execution models: `ExecuteCodeRequest` (`python_code`, `input_data`), `ExecuteCodeResponse` (`result`, `logs`, `execution_time_ms`, `memory_used_mb`).

## Step 2 – Tool implementation (sidecar agents/ module)

```
from pydantic import BaseModel, Field
from pydantic_ai import Agent
from pydantic_ai.models.openai import OpenAIModel

class CodegenRequest(BaseModel):
 prompt: str
 context: dict = Field(default_factory=dict)

class CodegenResponse(BaseModel):
 python_code: str
 summary: str
 allowed_imports: list[str] = Field(default=["numpy", "pandas", "math"])

class SqlQueryRequest(BaseModel):
 prompt: str
 available_columns: list[str]
 table_name: str = "uploaded"

class SqlQueryResponse(BaseModel):
 sql: str
 summary: str
 parameters: dict[str, str] = Field(default_factory=dict)
 estimated_rows: int | None = None

class ValidationResult(BaseModel):
 is_valid: bool
 issues: list[str]
 severity: str = "error" # error, warning, info

def build_codegen_agent(settings):
 """Agent that generates safe Python code for calculations."""
 model = OpenAIModel(
 model_name=settings.ai_model,
 base_url=settings.ai_base_url,
 api_key=settings.ai_api_key,
)

 return Agent(
 model=model,
 system_prompt=(
 "You are a careful code generator for numerical calculations. "
 "ONLY use these imports: numpy, pandas, math. "
 "Never use: os, subprocess, socket, requests, urllib, eval, exec, compile, open, file "
 "operations. "
 "Return executable Python that defines a function named 'calculate' taking a dict as input. "
 "Include docstring explaining the calculation."
),
 result_type=CodegenResponse,
)

def build_sql_agent(settings):
 """Agent that generates safe SELECT-only SQL for DuckDB."""
 model = OpenAIModel(
 model_name=settings.ai_model,
 base_url=settings.ai_base_url,
```

```

 api_key=settings.ai_api_key,
)

 return Agent(
 model=model,
 system_prompt=(
 "You produce read-only SQL for DuckDB against the 'uploaded' table. "
 "ONLY generate SELECT statements. "
 "NEVER use: DROP, DELETE, UPDATE, INSERT, CREATE, ALTER, TRUNCATE. "
 "Always add LIMIT 200 unless user specifies a smaller limit. "
 "Use only columns provided in the request. "
 "Use parameterized queries with $param syntax for user inputs."
),
 result_type=SqlQueryResponse,
)

def build_python_validator_agent(settings):
 """Agent that validates Python code for safety."""
 model = OpenAIModel(
 model_name=settings.ai_model,
 base_url=settings.ai_base_url,
 api_key=settings.ai_api_key,
)

 return Agent(
 model=model,
 system_prompt=(
 "Review Python code for safety violations. Flag any of these as INVALID:\n"
 "- Imports other than numpy, pandas, math\n"
 "- Use of eval, exec, compile, __import__\n"
 "- File operations: open, read, write\n"
 "- Network access: socket, requests, urllib, http\n"
 "- System calls: os, subprocess, sys.exit\n"
 "- Missing 'calculate' function definition\n"
 "Return is_valid=false if ANY violation found. List all issues."
),
 result_type=ValidationResult,
)

def build_sql_validator_agent(settings):
 """Agent that validates SQL for safety."""
 model = OpenAIModel(
 model_name=settings.ai_model,
 base_url=settings.ai_base_url,
 api_key=settings.ai_api_key,
)

 return Agent(
 model=model,
 system_prompt=(
 "Review SQL for safety violations. Flag as INVALID if:\n"
 "- Contains DDL: CREATE, DROP, ALTER, TRUNCATE\n"
 "- Contains DML: INSERT, UPDATE, DELETE\n"
 "- Does not start with SELECT\n"
 "- References tables other than 'uploaded'\n"
 "- Uses columns not in the allowed list\n"
 "- Missing LIMIT clause (require LIMIT <= 1000)\n"
 "Return is_valid=false if ANY violation found."
),
 result_type=ValidationResult,
)

async def refine_code(

```

```

codegen_agent: Agent,
validator_agent: Agent,
request: CodegenRequest,
max_attempts: int = 2
) -> CodegenResponse:
 """Generate code, validate, and request patches if invalid."""
 attempt_num = 0
 last_code = None

 while attempt_num <= max_attempts:
 # Generate code
 if attempt_num == 0:
 result = await codegen_agent.run(request.prompt, message_history=[])
 else:
 # Refine based on validation issues
 refine_prompt = (
 f"Fix this code to pass validation. Issues: {'', '.join(validation.issues)}\n"
 f"Original request: {request.prompt}\n"
 f"Previous code:\n{last_code}"
)
 result = await codegen_agent.run(refine_prompt, message_history=[])

 last_code = result.data.python_code

 # Validate
 validation = await validator_agent.run(
 f"Validate this code:\n{last_code}",
 message_history=[]
)

 if validation.data.is_valid:
 return result.data

 attempt_num += 1

 # All attempts exhausted
 raise ValueError(f"Validation failed after {max_attempts+1} attempts: {validation.data.issues}")

async def refine_sql(
 sql_agent: Agent,
 validator_agent: Agent,
 request: SqlQueryRequest,
 max_attempts: int = 2
) -> SqlQueryResponse:
 """Generate SQL, validate, and request patches if invalid."""
 attempt_num = 0
 last_sql = None

 while attempt_num <= max_attempts:
 if attempt_num == 0:
 prompt = f"{request.prompt}\nAvailable columns: {'', '.join(request.available_columns)}"
 result = await sql_agent.run(prompt, message_history=[])
 else:
 refine_prompt = (
 f"Fix this SQL to pass validation. Issues: {'', '.join(validation.issues)}\n"
 f"Original request: {request.prompt}\n"
 f"Available columns: {'', '.join(request.available_columns)}\n"
 f"Previous SQL:\n{last_sql}"
)
 result = await sql_agent.run(refine_prompt, message_history=[])

 last_sql = result.data.sql

 # Validate
 validation = await validator_agent.run(

```

```

 f"Validate this SQL:\n{last_sql}\nAllowed columns: {'', '.join(request.available_columns))",
 message_history=[]
)

 if validation.data.is_valid:
 return result.data

 attempt_num += 1

 raise ValueError(f"SQL validation failed after {max_attempts+1} attempts:
{validation.data.issues}")

```

### Step 3 – Sidecar FastAPI routes (description)

- `POST /ai/codegen` → runs codegen agent with refine loop, returns `CodegenResponse` . Accepts `max_attempts` param (default 2).
- `POST /ai/validate/python` → runs Python validator agent on `python_code` , returns `ValidationResult` .
- `POST /ai/validate/sql` → runs SQL validator agent on `sql` + `available_columns` , returns `ValidationResult` .
- `POST /ai/sql` → runs SQL agent with refine loop, returns `SqlQueryResponse` .
- `POST /ai/execute` → executes validated Python in sandboxed subprocess (see Security Hardening), returns `ExecuteCodeResponse` .
- Propagate `X-Trace-Id` , log agent name + attempt count + validation outcome + execution time.
- Error handling per matrix: 400 (bad request), 422 (validation fail), 500 (execution crash), 503 (LLM timeout).

### Step 4 – Calculator backend integration (description)

Add advanced endpoints to the calculator service:

- `POST /calculate/advanced` → accepts `CodegenRequest` , calls sidecar `/ai/codegen` with refine loop, then `/ai/validate/python` ; returns code + validation (no execution yet).
- `POST /calculate/advanced/execute` → accepts `ExecuteCodeRequest` (`python_code` + `input_data`), optionally revalidates, then calls `/ai/execute` with sandboxed limits.
- `POST /calculate/sql` → accepts `SqlQueryRequest` , calls sidecar `/ai/sql` with refine loop, then forwards SQL to DuckDB microservice `/query` with parameter binding.
- Return results with `X-Trace-Id` , execution metadata (`time_ms`, `memory_mb`, `attempts`), and validation summary.
- On validation failure after max attempts, return 422 with all `ValidationResult.issues` from final attempt.
- On execution timeout, return 500 with partial logs and timeout flag.

Example flow (Python):

```

Client → Calculator /calculate/advanced → Sidecar /ai/codegen (with refine loop) → LLM
 ↓
 Sidecar /ai/validate/python → LLM

Client → Calculator /calculate/advanced/execute → Sidecar /ai/execute (sandboxed) → Response

```

Example flow (SQL over CSV via DuckDB microservice):

```

Client → Calculator /calculate/sql → Sidecar /ai/sql (with refine loop) → LLM
 ↓
 Sidecar /ai/validate/sql → LLM

```

## Security Hardening

### Code Execution Sandboxing

#### Option 1 – RestrictedPython (in-process):

```
from RestrictedPython import compile_restricted, safe_globals
import resource

def execute_sandboxed(code: str, input_data: dict, timeout: int = 10) -> dict:
 # Compile with restrictions
 byte_code = compile_restricted(code, '<string>', 'exec')

 # Set resource limits
 resource.setrlimit(resource.RLIMIT_CPU, (timeout, timeout))
 resource.setrlimit(resource.RLIMIT_AS, (256 * 1024 * 1024, 256 * 1024 * 1024))

 # Execute with safe globals
 exec_globals = safe_globals.copy()
 exec_globals['__builtins__'] = {
 '__import__': __import__,
 '__build_class__': __build_class__,
 }
 exec(byte_code, exec_globals)

 # Call the calculate function
 return exec_globals['calculate'](input_data)
```

#### Option 2 – Docker container isolation:

```
import docker
import asyncio

async def execute_in_container(code: str, input_data: dict, timeout: int = 10) -> dict:
 client = docker.from_env()
 container = client.containers.run(
 "python:3.11-slim",
 command=["python", "-c", code],
 environment={"INPUT": json.dumps(input_data)},
 mem_limit="256m",
 cpu_period=100000,
 cpu_quota=50000, # 50% of one CPU
 network_disabled=True,
 detach=True,
)

 try:
 await asyncio.wait_for(container.wait(), timeout=timeout)
 logs = container.logs().decode()
 return json.loads(logs)
 finally:
 container.remove(force=True)
```

### Input Validation

- **CSV uploads:** Max file size 10MB; validate headers match expected schema; scan for zip bombs.

- **SQL parameters:** Type-check all parameters; reject overly long strings (>1000 chars).
- **Python code:** Max code length 5000 chars; syntax check with `ast.parse()` before LLM call.
- **Algorithmic complexity:** Reject code with nested loops >3 levels deep; limit list comprehension nesting.

## Rate Limiting

Implement per-endpoint rate limits:

- `/ai/codegen` : 10 requests/minute per IP
- `/ai/sql` : 20 requests/minute per IP
- `/ai/execute` : 5 requests/minute per IP (most resource-intensive)

## Tests (run locally)

- **Unit (sidecar):** `httpx.MockTransport` for deterministic Python and SQL responses; validator rejects unsafe imports/DDL and missing LIMIT; timeout/non-JSON tests.
- **Integration (calculator + sidecar + DuckDB):** `AsyncClient` against test app; `/calculate/advanced` returns code + validation; `/calculate/advanced/execute` runs sandboxed code; `/calculate/sql` returns rows; X-Trace-Id propagation; 422 on bad code/SQL.
- **Contract:** Assert schemas for `CodegenResponse`, `ValidationResult`, `SqlQueryResponse` stay in sync via `model_json_schema()`.
- **UI smoke:** Streamlit "Run Analysis" hits calculator-only endpoints; ensure no direct calls to sidecar/DuckDB from UI.

## Lab 2 – Connect real endpoints (45 min)

Goal: swap the mocked LLM for a real one (LM Studio/vLLM or Google AI Studio) and observe telemetry across Python and SQL flows.

### Local tracks (LM Studio or vLLM)

- Start the model server; confirm `GET /v1/models` works.
- Set `AI_BASE_URL` and `AI_API_KEY` (dummy for LM Studio) in `.env`.
- Hit `POST /calculate/advanced` with a prompt like "calculate mean and standard deviation of this list" and verify:
  - Generated code uses only numpy/pandas
  - Validation passes and refine loop returns best candidate
  - `X-Trace-Id` appears in logs from both services
- Then call `POST /calculate/advanced/execute` with the returned `python_code` and sample input; assert the sandboxed execution result and execution metadata.
- Hit `POST /calculate/sql` with "give me avg and max of sales by region" and verify:
  - SQL is SELECT-only, scoped to `uploaded` table, has `LIMIT`
  - DuckDB returns expected rows
- Test error scenarios:
  - Stop LLM server mid-request → verify 503 response
  - Send malformed prompt → verify graceful handling

## Google AI Studio track

- Export `GOOGLE_API_KEY` + `GOOGLE_GEMINI_MODEL` .
- Swap the agent model to `GoogleModel` :

```
from pydantic_ai.models.google import GoogleModel
agent = Agent(
 model=GoogleModel(model=settings.ai_model, api_key=settings.ai_api_key),
 system_prompt="You are a code generator for safe numerical calculations. Only use numpy, pandas, and math.",
 tools=[codegen_tools(settings)],
 output_type=CodegenResponse,
)
```

- Repeat the `/calculate/advanced` , `/calculate/advanced/execute` , and `/calculate/sql` calls with the same test cases; observe latency/quotas and logtrace.
- Compare response times: local vLLM vs Google AI Studio (document in README).

## Observability checkpoints

- Verify structured logs include:
  - LLM model name and provider
  - Token usage (if available from LLM response)
  - Request/response latency (P50, P95, P99)
  - Validation pass/fail rates by agent type
  - Refine loop: attempts distribution, convergence rate
  - Code execution time and memory usage
  - SQL path: rejected vs executed SQL count, table/column allowlist hits, DuckDB query time
  - Error rates by type (timeout, validation, execution)
- Set up alerts for:
  - Validation failure rate >20% (possible prompt drift)
  - LLM timeouts >5% of requests
  - Execution errors >10% (code generation quality issue)
  - SQL rejection spikes (>50% rejected) or DDL attempts detected
  - Memory usage >80% of limit
  - Request queue depth >50

## Streamlit Integration

Add these components to the Streamlit UI:

### CSV Upload & Analysis Tab (frontend talks only to calculator backend):

```
uploaded_file = st.file_uploader("Upload CSV", type=["csv"])
if uploaded_file:
 # Upload via calculator backend (which forwards to DuckDB microservice)
 response = httpx.post(f"{CALCULATOR_URL}/data/upload", files={"file": uploaded_file})
 payload = response.json()
 columns = payload["columns"]
 st.success(f"Uploaded {payload['rows']} rows")

 # Free-text SQL query
 query = st.text_area("What do you want to know?",
```

```

 "Show me average sales by region")
if st.button("Run Analysis"):
 with st.spinner("Generating SQL..."):
 result = httpx.post(f"{CALCULATOR_URL}/calculate/sql", json={
 "prompt": query,
 "available_columns": columns
 })
 st.code(result.json()["sql"], language="sql")
 st.dataframe(result.json()["rows"])

```

### Custom Calculation Tab:

```

calc_prompt = st.text_area("Describe calculation",
 "Calculate compound annual growth rate")
if st.button("Generate Code"):
 with st.spinner("Generating and validating code..."):
 result = httpx.post(f"{CALCULATOR_URL}/calculate/advanced", json={
 "prompt": calc_prompt
 })

 st.subheader("Generated Code")
 st.code(result.json()["python_code"], language="python")

 st.subheader("Validation")
 if result.json()["validation"]["is_valid"]:
 st.success("✓ Code passed safety checks")
 else:
 st.error("x Validation issues: " + ", ".join(result.json()["validation"]["issues"]))

if st.button("Execute"):
 exec_result = httpx.post(f"{CALCULATOR_URL}/calculate/advanced/execute",
 json={
 "python_code": result.json()["python_code"],
 "input_data": {}
 })
 st.json(exec_result.json()["result"])

```

## Wrap-Up & Success Criteria

- ☐ Frontend calls only the calculator backend; calculator proxies sidecar and DuckDB.
- ☐ `/calculate/advanced` returns generated + validated code; `/calculate/advanced/execute` runs it in the sandbox with limits (refine loop capped).
- ☐ `/calculate/sql` returns SELECT-only SQL + rows from DuckDB; blocks DDL/DML.
- ☐ `/ai/codegen`, `/ai/validate/python`, `/ai/validate/sql`, `/ai/sql`, `/ai/execute` return typed payloads and log X-Trace-Id.
- ☐ Tests cover sidecar units, calculator-side integration, schema contracts, and UI smoke.
- ☐ README/changelog note AI usage, env vars, telemetry defaults, and safety rails.
- ☐ Local and remote LLM paths both exercised (document latency/quotas).

## Troubleshooting

- **LLM returns non-JSON** → enforce `response_format`; wrap parse in try/except and return 422 with details.
- **403/401 from Google** → confirm `GOOGLE_API_KEY`, `GOOGLE_GEMINI_MODEL`; check quotas.
- **Local model unreachable** → `curl ${AI_BASE_URL}/v1/models`; restart LM Studio/vLLM; check firewall.
- **pytest hits real network** → use MockTransport; set `AI_API_KEY=mock`; override clients in tests.

- **Validation misses unsafe code/SQL** → tighten prompts; add regex/AST guards; reduce allowed imports/columns.
- **Code execution hangs** → lower timeout; enforce memory/CPU limits; prefer container sandbox for risky code.
- **UI bypasses backend** → audit Streamlit calls; ensure only calculator URLs are used.
- **SQL rejects valid queries** → ensure column list matches DuckDB schema; adjust default LIMIT; log rejected SQL for tuning.

# Session 09 – Async Recommendation Refresh (EX3 Reliability Checkpoint)

docs/sessions/session-09.md

- **Date:** Monday, 11/05/2026
- **Theme:** Move the recommendation refresh pipeline to asynchronous execution, add retries/backoff, and guard against duplicate work with idempotency keys so every EX3 project ships a proven background job + telemetry loop.

## Learning Objectives

- Call FastAPI endpoints with `httpx.AsyncClient` using Asynchronous Server Gateway Interface (ASGI) transport for in-process tests.
- Introduce bounded concurrency with `asyncio.Semaphore` and implement retry/backoff policies ( `anyio` , `tenacity` ).
- Add idempotency keys to POST requests to avoid double-processing and design resilience tests around them.
- Instrument async flows with trace identifiers (IDs) and metrics hooks for future observability.
- Produce the EX3-required `scripts/refresh.py` Typer command plus pytest coverage that proves the async path works end to end.

## Session Focus

1. Ship a bounded async refresher with retries/backoff and per-job idempotency keys.
2. Expose the refresher via `scripts/refresh.py` and cover it with at least one `pytest.mark.anyio` test that hits FastAPI through ASGI transport.
3. Capture a telemetry snippet (Logfire span or log excerpt) that proves `X-Trace-Id` + `Idempotency-Key` headers survive the full path and paste it into `docs/EX3-notes.md` .

## Before Class – Async Preflight (Just-in-Time Teaching, JiTT)

- Install async tooling:

```
uv add httpx anyio tenacity
```

- Add telemetry + cache helpers you will lean on later:

```
uv add logfire redis
```

- Review Python's `asyncio` basics (Learning Management System (LMS) primer) and jot one question about concurrency hazards.
- Ensure Exercise 3 (EX3) materials are cloned and the local API/interface run cleanly; if you already support Docker Compose locally, make sure `docker compose up` succeeds so you can reuse the same stack for async drills.

 **EX3 Deliverable:** By the end of this session every team must check in (1) `scripts/refresh.py` with bounded concurrency + retries, (2) at least one `pytest.mark.anyio` test that exercises the refresher against the FastAPI app via ASGI transport, and (3) a short log excerpt (paste into `docs/EX3-notes.md` ) showing `Idempotency-Key` and `X-Trace-Id` headers working together. Later worker/Redis labs assume this script exists.

## Simple Demo – Async refresher + telemetry loop

1. Start the FastAPI app ( `uv run uvicorn app.main:app --reload` ) so the refresher has a live target.
2. Run `uv run python scripts/refresh.py run --limit 3` and watch each POST log its `X-Trace-Id` + `Idempotency-Key` .
3. Paste the snippet into `docs/EX3-notes.md` and rerun the command to confirm duplicate keys short-circuit instead of inserting twice.

## Agenda

Segment	Duration	Format	Focus
Check-in & EX3 kickoff	10 min	Discussion	Share Exercise 3 (EX3) project scope and reliability concerns.
Async primer	18 min	Talk + whiteboard	Event loop, async/await, cooperative multitasking, pitfalls.
Micro demo: AsyncClient + ASGI transport	5 min	Live demo	Call FastAPI without network via <code>httpx.ASGITransport</code> .
Reliability patterns	12 min	Talk	Retries, circuit breakers, idempotency keys, tracing.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided coding</b>	<b>Async recommendation job with bounded concurrency + retries.</b>
Break	10 min	—	Launch the shared <a href="#">10-minute timer</a> .
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided testing</b>	<b>Async tests, idempotency guarantees, instrumentation.</b>
Wrap-up	10 min	Discussion	Next steps for Exercise 3 (EX3) milestone, logging notes, Redis preview.

## Part A – Theory Highlights

1. **Event loop refresher:** tasks share a thread, await input/output (I/O), avoid CPU-heavy work. Show `asyncio.create_task` (schedule work without blocking), `asyncio.gather` (wait for many coroutines at once), and `asyncio.Semaphore` (cap concurrency) so every student knows the building blocks.
2. **Retry/backoff:** exponential vs. jitter, idempotent vs. non-idempotent operations, using `tenacity` decorators.
3. **Idempotency keys:** Accept `Idempotency-Key` header, store processed keys (in-memory or Redis), and short-circuit duplicates.
4. **Instrumentation:** Keep `X-Trace-Id` consistent; plan to emit metrics using Prometheus/Redis.

## Motivating Micro Demos (5 minutes each)

Give students a quick taste of why we pair async work with Redis + Logfire before diving into the heavier lab steps.

1. **Redis quick win:**

```
docker run --rm -p 6379:6379 redis:7-alpine
redis-cli set "recommend:pending" 2
redis-cli decr "recommend:pending"
redis-cli get "recommend:pending"
```

Explain that the same counter powers your retry/idempotency checks once the async refresher fans out jobs. Seeing the number drop in real time makes the need for a shared cache tangible.

## 2. Logfire heartbeat: (reuse the existing project virtualenv)

```

uv run python - <<'PY'
import asyncio
import logfire

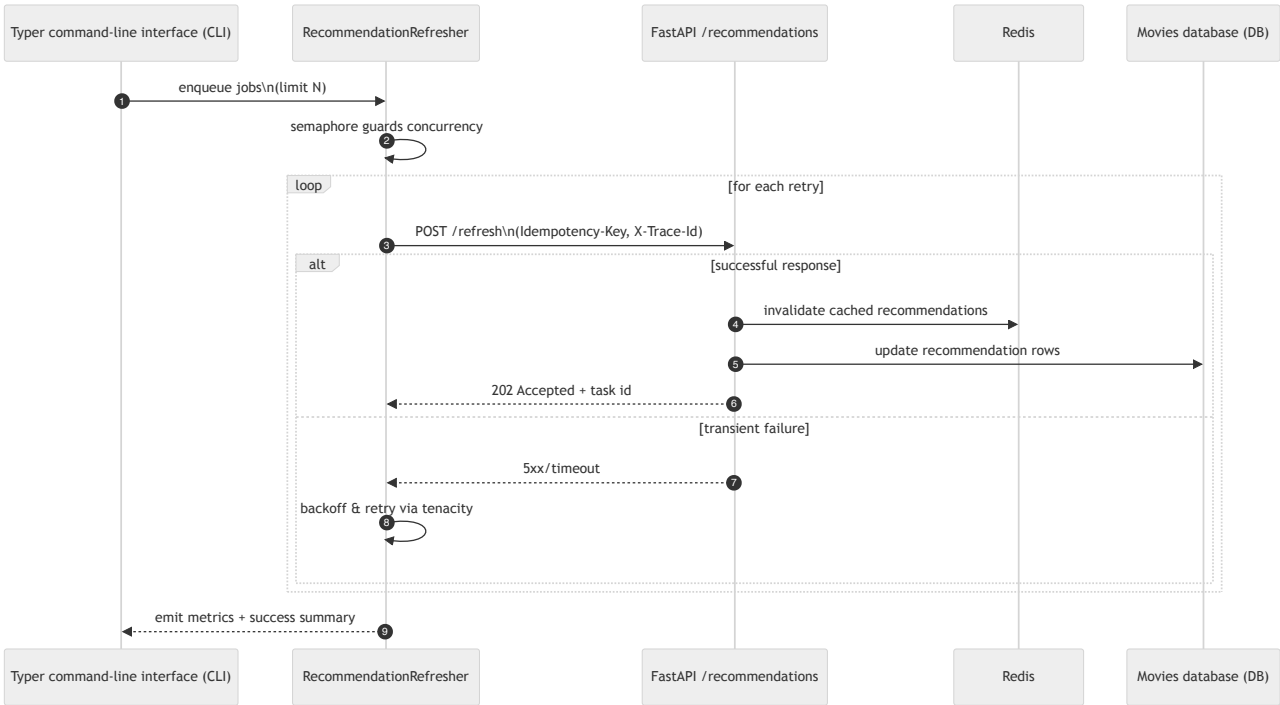
logfire.configure()

async def demo_task(name: str):
 with logfire.span("demo.refresh", task=name):
 await asyncio.sleep(0.2)
 logfire.info("task-complete", task=name)

asyncio.run(asyncio.gather(demo_task("a"), demo_task("b")))
PY

```

Point out the spans and structured logs in the console—Session 09’s refresher will emit the same telemetry so later exercises have trace data to reference.



## Part B – Lab 1 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Review refresher settings ( `refresh_max_concurrency` , `trace_id` ).
- **Minutes 10–25** – Implement async refresher with semaphore + retries.
- **Minutes 25–35** – Add idempotency keys and confirm with logs.
- **Minutes 35–45** – Trigger jobs via Typer CLI and inspect trace IDs.

## 1. Async recommendation service ( app/recommendation.py )

```
from __future__ import annotations

import asyncio
from dataclasses import dataclass
from typing import Iterable

import httpx
import logfire
from tenacity import AsyncRetrying, retry_if_exception_type, stop_after_attempt,
wait_exponential_jitter

from app.config import Settings

@dataclass
class RecommendationJob:
 movie_id: int
 payload: dict[str, int]

class RecommendationRefresher:
 def __init__(self, settings: Settings) -> None:
 self.settings = settings
 self.client = httpx.AsyncClient(base_url=settings.api_base_url, timeout=10.0)
 self._semaphore = asyncio.Semaphore(settings.refresh_max_concurrency)
 self._logger = logfire.with_fields(component="recommendation_refresher")

 async def refresh(self, jobs: Iterable[RecommendationJob]) -> None:
 job_list = list(jobs)
 with self._logger.span("recommendation.refresh", jobs=len(job_list)):
 tasks = [self._bounded_refresh(job) for job in job_list]
 await asyncio.gather(*tasks)

 async def _bounded_refresh(self, job: RecommendationJob) -> None:
 async with self._semaphore:
 async for attempt in AsyncRetrying(
 stop=stop_after_attempt(3),
 wait=wait_exponential_jitter(initial=0.5, max=5.0),
 retry=retry_if_exception_type(httpx.HTTPError),
):
 with attempt:
 await self._send_job(job)

 async def _send_job(self, job: RecommendationJob) -> None:
 idempotency_key = f"recommend:{job.movie_id}:{job.payload['user_id']}"
 with self._logger.span("recommendation.job", movie_id=job.movie_id):
 response = await self.client.post(
 "/recommendations/refresh",
 json=job.payload,
 headers={
 "X-Trace-Id": self.settings.trace_id,
 "Idempotency-Key": idempotency_key,
 },
)
 response.raise_for_status()
 logfire.info(
 "refresh-complete",
 movie_id=job.movie_id,
 status=response.status_code,
 idempotency_key=idempotency_key,
)
```

Track `refresh_max_concurrency` and `trace_id` via `Settings` (add defaults to config with environment overrides).

Update `app/config.py` (from Session 03) with async-specific settings:

```
class Settings(BaseSettings):
 # existing fields...
 api_base_url: str = "http://localhost:8000"
 refresh_max_concurrency: int = 4
 trace_id: str = "recommend-refresh"
```

Call `logfire.configure()` once during app or CLI startup (Session 07 introduced this pattern) so the spans and counters above stream to your local console or configured backend.

## 2. Async CLI trigger (`scripts/refresh.py`)

```
import asyncio
import typer

from app.config import Settings
from app.recommendation import RecommendationJob, RecommendationRefresher

app = typer.Typer(help="Async recommendation refresh")

@app.command()
def run(limit: int = 10) -> None:
 settings = Settings()
 jobs = [RecommendationJob(movie_id=i, payload={"movie_id": i, "user_id": 1}) for i in range(limit)]
 refresher = RecommendationRefresher(settings)
 asyncio.run(refresher.refresh(jobs))

if __name__ == "__main__":
 app()
```

Show logs with trace IDs and idempotency keys for each request.

 **Quick win:** Running `uv run python scripts/refresh.py run --limit 3` without duplicate inserts proves your idempotency guard is effective.

## Part C – Lab 2 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Mock async client with `ASGITransport` for in-process testing.
- **Minutes 10–25** – Assert idempotency behavior and retry logic.
- **Minutes 25–35** – Add caching hooks (Redis) and metrics counters.
- **Minutes 35–45** – Inject failures, inspect Logfire counters, and document learnings.

### 1. Async tests with `ASGITransport`

```
import asyncio

import httpx
import pytest
```

```

from app.main import app
from app.recommendation import RecommendationJob, RecommendationRefresher
from app.config import Settings

@pytest.mark.anyio
async def test_refresh_hits_endpoint(monkeypatch):
 transport = httpx.ASGITransport(app=app)
 client = httpx.AsyncClient(transport=transport, base_url="http://testserver")

 monkeypatch.setattr("app.recommendation.httpx.AsyncClient", lambda *args, **kwargs: client)

 settings = Settings(refresh_max_concurrency=2, api_base_url="http://testserver", trace_id="async-test")
 refresher = RecommendationRefresher(settings)

 job = RecommendationJob(movie_id=1, payload={"movie_id": 1, "user_id": 42})
 await refresher.refresh([job])

```

Explain `pytest.mark.anyio` and why we re-use ASGI transport to avoid network calls.

## 2. Idempotency guard test

Design FastAPI endpoint storing processed keys (in-memory for now). Write test to send duplicate job and assert second response returns 202 or similar without duplicate work.

## 3. Reliability metrics (stretch)

- Emit `logfire.counter("recommendation.retry", ...)` inside retry loop.
- Record durations with `logfire.timer()` context manager for future dashboards.

## 4. Failure injection

Use `pytest` to monkeypatch the endpoint and raise HTTP 500 on first attempt, confirm retry hits and eventual success.

## 5. Redis caching for recommendations

```

import json
import os
import redis

r = redis.from_url(os.getenv("REDIS_URL", "redis://localhost:6379/0"))

def cache_recommendations(user_id: int, recs: list[int]) -> None:
 key = f"recs:{user_id}"
 r.setex(key, 3600, json.dumps(recs))

def get_cached_recommendations(user_id: int) -> list[int] | None:
 key = f"recs:{user_id}"
 data = r.get(key)
 return json.loads(data) if data else None

@app.get("/recommendations/{user_id}")
async def recommend(user_id: int) -> dict[str, object]:
 if cached := get_cached_recommendations(user_id):
 return {"source": "cache", "recommendations": cached}

```

```
recs = await generate_recommendations(user_id)
cache_recommendations(user_id, recs)
return {"source": "fresh", "recommendations": recs}
```

Discuss time to live (TTL) strategy (one hour here) and note how a Redis deployment makes this production-ready.

## Wrap-up & Next Steps

-  Async refresher, retries with jitter, idempotency keys, async tests.
- Capture a Logfire screenshot or log excerpt plus Redis counter snippet in `docs/EX3-notes.md` so graders can trace your background jobs later.
- Confirm Redis is installed locally ( `brew install redis` or `docker run redis` ) so caching/idempotency checks are easy to repeat.

## Troubleshooting

- **RuntimeError: Event loop is closed** → avoid nested `asyncio.run`; use `pytest.mark.anyio` and `asyncio.get_event_loop_policy().new_event_loop()` if needed.
- **Idempotency store resets** → persist keys in Redis or file for long-running jobs.
- **Un-awaited coroutine warnings** → ensure every async call is awaited; use `pytest.raises` with async context for exceptions.

## Common pitfalls

- **Semaphore never releases** – wrap critical sections in `async with` to guarantee release even when exceptions occur.
- **Idempotency keys collide** – include both `movie_id` and `user_id` (or timestamp bucket) to avoid overwriting legitimate requests.
- **Redis connection refused** – ensure local Redis is running ( `brew services start redis` or Docker container) before enabling caching.

## Student Success Criteria

By the end of Session 09, every student should be able to:

- ☐ Refresh recommendations asynchronously with bounded concurrency and retries.
- ☐ Write async pytest suites using `httpx.AsyncClient` / `pytest.mark.anyio`.
- ☐ Cache recommendation results in Redis with idempotency keys to prevent duplicate work.
- ☐ Emit Logfire spans/counters that prove when refresh jobs run and succeed (screenshot or log excerpt stored in `docs/EX3-notes.md`).

If any box stays unchecked, book an async lab session soon after this one.

## AI Prompt Seeds

- “Write an async refresher that batches POST requests with bounded concurrency and retries using tenacity.”
- “Show how to use `httpx.AsyncClient` with `ASGITransport` in tests.”
- “Design an idempotency key strategy for POST `/recommendations/refresh` with sample tests.”

- “Instrument the refresher with Logfire spans and counters so each job logs status + idempotency key.”

# Session 10 – Docker Compose, Redis, and Service Contracts (EX3 Runbook Core)

docs/sessions/session-10.md

- **Date:** Monday, 18/05/2026
- **Theme:** Run the movie API with Redis and worker services via Docker Compose, enforce service contracts, and introduce rate limiting/background jobs so the EX3 stack has a single `docker compose up` story plus CI gates.

## Learning Objectives

- Model multi-service environments with Docker Compose (application programming interface (API), Redis, worker, proxy) and treat the compose file as the default EX3 launch command.
- Use Redis for caching, idempotency tracking, and rate limiting ( `slowapi` ) while wiring Session 09's async refresher into a background worker.
- Harden API contracts with Schemathesis (from Session 02 stretch) and document expectations in `docs/service-contract.md` .
- Wire coverage + contract tests into GitHub Actions for continuous integration (CI) confidence and capture the exact steps inside `docs/runbooks/compose.md` .

## Session Focus

1. Maintain a single `docker compose up` workflow that boots the API, Redis, and async worker with healthchecks.
2. Use Redis for rate limiting + caching (reusing Session 09's refresher) and explain the behavior inside `docs/service-contract.md` .
3. Lock in contract/CI coverage by showing how Schemathesis, pytest (with coverage), and the Compose runbook all tie together.

## Before Class – Compose Preflight (Just-in-Time Teaching, JiTT)

- Install Redis locally or ensure Docker Desktop can pull `redis:7-alpine` :

```
docker pull redis:7-alpine
```
- For hands-on Compose drills, open Docker Desktop and launch a **Labspace** from the left navigation (blue beaker icon). The **Building agentic apps with Docker** and **Container-supported development** lab cards mirror Session 10's multi-service stack, so you can rehearse Compose + Redis workflows even if your laptop cannot run the full stack.
- Review the provided Compose primer (Learning Management System (LMS)) and list one question about networking or environment variables.
- Run `uv run schemathesis run docs/contracts/openapi.json --checks status_code_conformance --dry-run` to confirm tooling works.
- Copy Session 09's `scripts/refresh.py` into your repo (if you coded along elsewhere); Session 10 extends it into the worker container.



**EX3 Deliverable:** Ship a working `compose.yaml` with services for API, Redis, and the async worker plus CI instructions ( `docs/runbooks/compose.md` ) that describe `docker compose up` , contract testing, and how to view rate-limit headers. Instructors will grade EX3 by running these exact commands.

## Simple Demo – Compose stack in one command

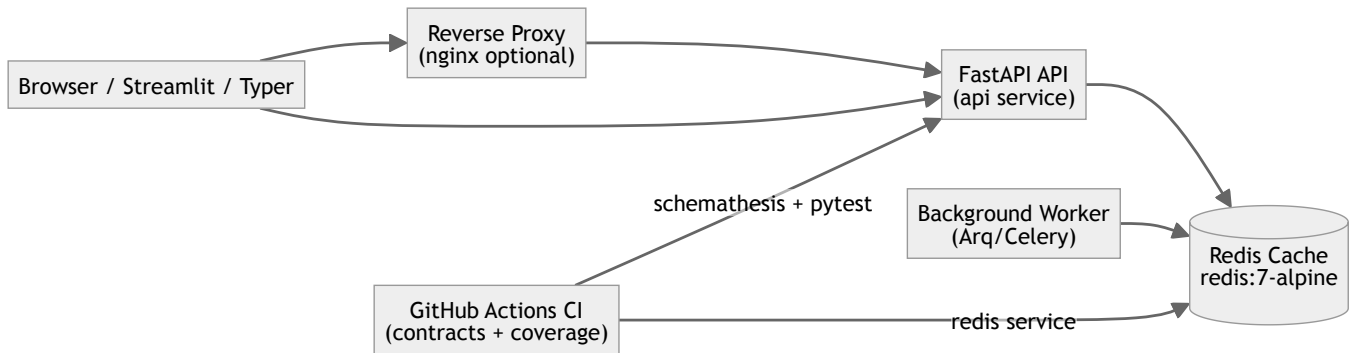
1. From the repo root, run `docker compose up --build` and wait until API, worker, and Redis report `healthy`.
2. In another terminal, hit `curl -i http://localhost:8000/movies` twice—confirm the first response returns JSON plus `X-RateLimit-*` headers and the second hits Redis cache (latency drop in logs).
3. Run `redis-cli -u redis://localhost:6379/0 keys 'rate:*` (or `docker exec` if using Compose) to prove rate-limit counters exist, then stop the stack with `Ctrl+C`.

## Agenda

Segment	Duration	Format	Focus
Recap & async wins	10 min	Discussion	Share Session 09 takeaways, identify bottlenecks.
Compose architecture	18 min	Talk + diagram	api + redis + worker + proxy, networks, volumes.
Micro demo: redis cache hit	5 min	Live demo	Use <code>redis-cli</code> to show cache set/get vs API latency.
Contract testing refresher	12 min	Talk	Schemathesis, JSON schema examples, service obligations.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided build</b>	<b>Docker Compose stack with FastAPI, Redis, background worker.</b>
Break	10 min	—	Launch the shared <a href="#">10-minute timer</a> .
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided validation</b>	<b>Rate limiting, contract tests, GitHub Actions CI pipeline.</b>
Wrap-up	10 min	Questions and Answers (Q&A)	Exercise 3 (EX3) milestone reminder, deployment prep.

## Part A – Theory Highlights

1. **Compose structure:** `services` block, named volumes, networks, environment variables, healthchecks ( `depends_on` with `service_healthy` ).
2. **Redis roles:** caching (movie list), rate limiting tokens, job queue backend—make sure students know Redis is an in-memory key-value datastore that excels at fast counters and lists.
3. **Background jobs:** choose one lightweight runner (Arq or Celery). For class we'll show Arq because it rides on asyncio.
4. **Contracts:** `docs/service-contract.md` documents request/response shapes; Schemathesis enforces them alongside integration tests.
5. **Continuous integration (CI) pipeline:** Compose for local dev, GitHub Actions for CI (use `services.redis` job with `services.redis`, run `uv sync --frozen`, `pytest --cov`, `schemathesis` ).



## Part B – Hands-on Lab 1 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Scaffold `compose.yaml` and review service relationships.
- **Minutes 10–25** – Add cache + rate limit environment variables and healthchecks.
- **Minutes 25–35** – Launch the stack ( `docker compose up` ) and verify health endpoints.
- **Minutes 35–45** – Wire Arq worker + caching helper and explore logs.

### 1. Compose file ( `compose.yaml` )

```

services:
 api:
 build: .
 command: uv run uvicorn app.main:app --host 0.0.0.0 --port 8000
 ports:
 - "8000:8000"
 environment:
 - MOVIE_REDIS_URL=redis://redis:6379/0
 - MOVIE_RATE_LIMIT_PER_MINUTE=20
 depends_on:
 redis:
 condition: service_healthy
 healthcheck:
 test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
 interval: 30s
 timeout: 5s
 retries: 3

 worker:
 build: .
 command: uv run arq app.worker.WorkerSettings
 environment:
 - MOVIE_REDIS_URL=redis://redis:6379/0
 depends_on:
 redis:
 condition: service_healthy

 redis:
 image: redis:7-alpine
 ports:
 - "6379:6379"
 healthcheck:
 test: ["CMD", "redis-cli", "ping"]
 interval: 10s

```

```
timeout: 3s
retries: 5
```

Explain networks (default) and volumes (add later for persistence if needed).

Spin everything up:

```
docker compose up --build
```

Keep the stack running while you exercise the API and worker.



**Quick win:** When `docker compose ps` shows healthy API/worker/redis services, you're operating a multi-service stack locally.

## 2. Redis client helper ( `app/cache.py` )

```
from functools import lru_cache

import redis.asyncio as redis

from app.config import Settings

@lru_cache(maxsize=1)
def get_redis(settings: Settings) -> redis.Redis:
 return redis.from_url(settings.redis_url, decode_responses=True)
```

Extend `Settings` with `redis_url: str` and `rate_limit_per_minute: int`.

## 3. Rate limiting middleware ( `app/rate_limit.py` )

```
import time

from fastapi import HTTPException, Request

from app.cache import get_redis
from app.config import Settings

async def rate_limit(request: Request, call_next):
 settings = Settings()
 redis_client = get_redis(settings)
 key = f"rate:{request.client.host}:{request.url.path}"
 window = 60
 max_requests = settings.rate_limit_per_minute

 current = await redis_client.incr(key)
 if current == 1:
 await redis_client.expire(key, window)

 if current > max_requests:
 raise HTTPException(status_code=429, detail="Too many requests")

 response = await call_next(request)
 response.headers["X-RateLimit-Limit"] = str(max_requests)
 response.headers["X-RateLimit-Remaining"] = str(max_requests - current)
 return response
```

Mount middleware in `app/main.py` before routes.

## 4. Background worker ( `app/worker.py` )

```
from arq import cron
from arq.connections import RedisSettings

from app.config import Settings

class WorkerSettings:
 redis_settings = RedisSettings.from_url(Settings().redis_url)

 async def startup(self, ctx):
 ctx["settings"] = Settings()

 async def shutdown(self, ctx):
 ctx.pop("settings", None)

 async def refresh_recommendations(self, ctx, user_id: int) -> None:
 settings: Settings = ctx["settings"]
 # call async refresher from Session 09

 functions = [refresh_recommendations]
 cron_jobs = [cron(refresh_recommendations, cron_string="0 * * * *", kwargs={"user_id": 42})]
```

Explain Arq's simplicity: it is a lightweight task queue that reuses `async def` functions and the same Redis instance, so students do not need to learn a new worker process model. Contrast it briefly with **Celery**, the older, feature-rich queue that supports multiple brokers (Redis/RabbitMQ) but demands more setup—point out when the heavier tool is worth the trade-off.

## 5. Caching GET `/movies`

Wrap the list route in `app/main.py` :

```
import json

from fastapi import Depends

from app.cache import get_redis

@app.get("/movies", response_model=list[Movie])
async def list_movies(repository: RepositoryDep, settings: SettingsDep) -> list[Movie]:
 redis_client = get_redis(settings)
 cache_key = "movies:list"
 cached = await redis_client.get(cache_key)
 if cached:
 return json.loads(cached)

 result = [movie for movie in repository.list()]
 await redis_client.setex(cache_key, 60, json.dumps([movie.model_dump() for movie in result]))
 return result
```

Remember to invalidate cache in POST/PUT/DELETE handlers.

## Part C – Lab 2 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Run Schemathesis against the live Compose stack.
- **Minutes 10–25** – Add GitHub Actions job steps for tests + contract checks.
- **Minutes 25–35** – Implement rate limiting tests (429) and headers.
- **Minutes 35–45** – Update service contract docs and capture artifacts for EX3.

### 1. Contract tests with Schemathesis

```
uv run schemathesis run http://localhost:8000/openapi.json --checks status_code_conformance --workers 2
```

Address failures immediately; update docs if behavior changes. Store command in `Makefile` or `scripts/check_contract.sh`.

### 2. GitHub Actions pipeline

`.github/workflows/ci.yaml` (excerpt):

```
jobs:
 test:
 runs-on: ubuntu-latest
 services:
 redis:
 image: redis:7-alpine
 ports:
 - 6379:6379
 steps:
 - uses: actions/checkout@v4
 - uses: astral-sh/setup-uv@v1
 - run: uv sync --frozen
 - run: uv run pytest --cov=app --cov-report=term-missing
 - run: uv run schemathesis run http://localhost:8000/openapi.json --checks
 status_code_conformance --dry-run
```

Explain how to swap the dry run for real execution once you can boot the API inside CI (Compose + `uvicorn` background run).

### 3. Rate limiting test (429)

```
def test_rate_limit(client):
 for _ in range(Settings().rate_limit_per_minute):
 assert client.get("/movies").status_code == 200

 response = client.get("/movies")
 assert response.status_code == 429
 assert response.json()["detail"] == "Too many requests"
```

Use `freezegun` or `monkeypatch` time if the window needs reset.

## 4. Contract documentation

Update `docs/service-contract.md` with:

- Rate limit headers and expected values.
- Cache invalidation rules.
- Background job service level agreement (SLA) (refresh runs every hour; immediate manual trigger via CLI).

## Wrap-up & Next Steps

---

-  Compose stack, Redis cache + rate limiting, background worker, contract testing, CI pipeline outline.
- Audit secrets, rotate tokens, and create `.env.example` entries for Redis auth if enabled.
- Point teams to the EX3 brief in [docs/exercises.md](#) so they align infrastructure deliverables with the milestone.

## Troubleshooting

---

- **Redis connection refused** → confirm Compose network is up or local Redis running; check `redis_url` env.
- **Slow Schemathesis runs** → narrow to critical endpoints with `--endpoint` filter or run in CI nightly.
- **Arq import error** → install `uv add arq`; ensure worker service uses same image/tag as API.

## Common pitfalls

- **Compose race conditions** – use `depends_on` with `condition: service_healthy` to avoid worker failures when Redis isn't ready.
- **Environment drift** – keep `.env` defaults aligned with `compose.yaml` to prevent mismatched URLs locally vs. CI.
- **Schemathesis hitting old schema** – regenerate `openapi.json` after code changes or mount the live URL instead of static file.

## Student Success Criteria

---

By the end of Session 10, every student should be able to:

- ☐ Run the FastAPI API, Redis, worker, and proxy together via Docker Compose.
- ☐ Enforce rate limits and cache hits backed by Redis inside the stack.
- ☐ Execute contract tests (Schemathesis) and CI-ready pipelines covering the multi-service environment.

**If any item lags, assign a small-group infra session to close the gap.**

## AI Prompt Seeds

---

- “Draft a docker-compose.yml with FastAPI, Redis, and an Arq worker including healthchecks.”
- “Write FastAPI middleware that rate limits requests using Redis tokens.”
- “Generate a GitHub Actions workflow that runs pytest with coverage and Schemathesis contract tests.”

# Session 11 – Security Foundations (EX3 Baseline Controls)

docs/sessions/session-11.md

- **Date:** Monday, 25/05/2026
- **Theme:** Secure the movie service with proper authentication (auth), secrets hygiene, and threat modeling so EX3 submissions include hashed credentials, JWT-protected routes, and documented secret rotation.

## Learning Objectives

- Hash passwords with `passlib[bcrypt]`, issue JSON Web Tokens (JWTs) (`pyjwt`) with expiration, issuer, and audience claims.
- Apply Open Worldwide Application Security Project (OWASP) API Top 3 mitigations: authentication, input validation, secret management.
- Audit `.env.example`, rotate keys, and document secure defaults for Docker/Compose plus GitHub Actions.
- Write security-focused tests (401/403 paths, JWT expiry, secret exposure checks) and capture the results in the EX3 runbook checklist.

## Session Focus

1. Replace plaintext credentials with bcrypt hashes and centralize JWT settings/rotation guidance.
2. Gate at least one route with role-aware JWT checks plus tests that cover 401/403/expiry paths.
3. Record the hardening steps (hashing, JWT config, secret scanning) inside `docs/EX3-notes.md` so graders can trace the controls.

## Before Class – Security Preflight (Just-in-Time Teaching, JiTT)

- Install dependencies:

```
uv add passlib[bcrypt] pyjwt
```

- Rotate any hard-coded secrets in your repo; ensure `.env` is gitignored and `.env.example` is up to date.
- Review OWASP API Security Top 10 summary (Learning Management System (LMS)) and bring one question.
- Optional but encouraged: scan your repo with `trufflehog filesystem --exclude .git` to confirm secrets hygiene.



**EX3 Deliverable:** Every team must (1) protect at least one API route with JWT-based role checks, (2) document password hashing + secret rotation steps inside `docs/EX3-notes.md`, and (3) add tests that fail if a token is expired or missing the right role. These items sit on the EX3 grading checklist, so finish them during this session.

## Simple Demo – Login + protected endpoint

1. Start the API locally (`uv run uvicorn app.main:app --reload`) so `/token` and the protected route are reachable.
2. Request a token:

```
TOKEN=$(curl -s -X POST http://localhost:8000/token -d "username=teacher&password=classroom" -H "Content-Type: application/x-www-form-urlencoded" | jq -r '.access_token')
```

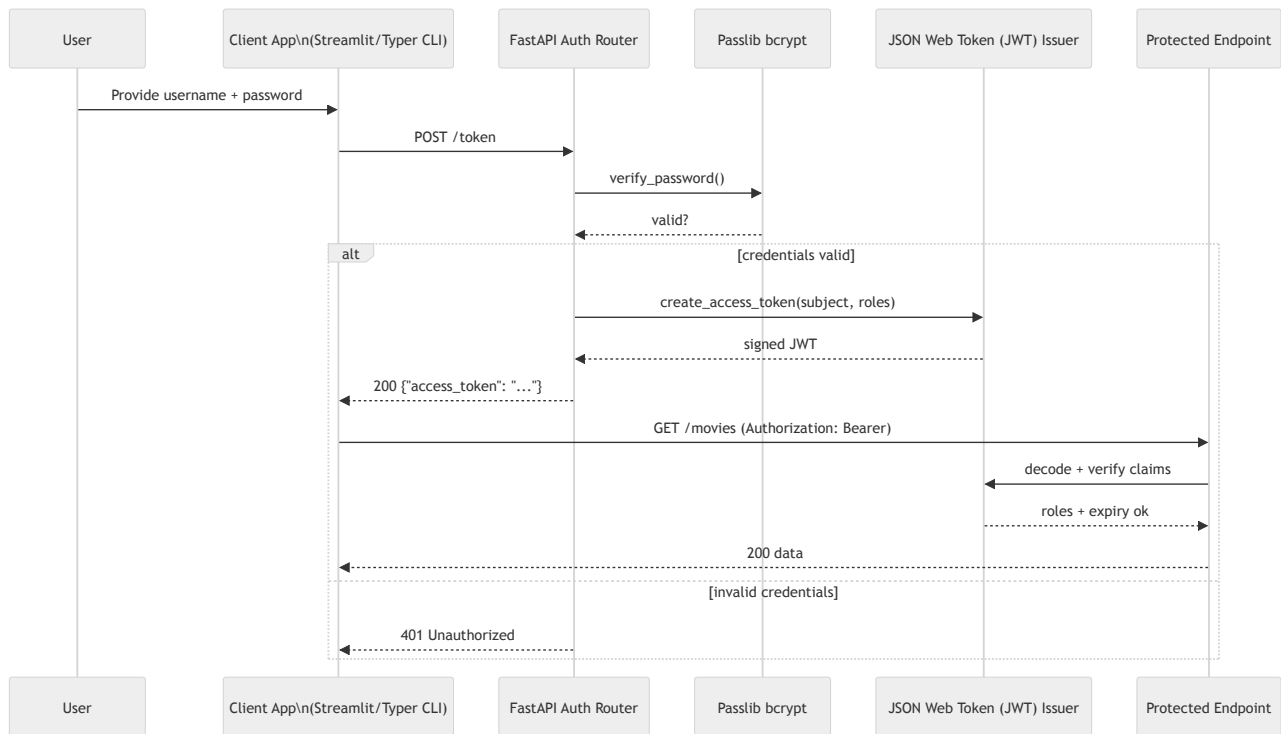
3. Use the token to hit the protected endpoint ( `curl -H "Authorization: Bearer $TOKEN" http://localhost:8000/movies` ) and rerun the same request without the header to confirm it returns 401/403.

## Agenda

Segment	Duration	Format	Focus
Threat model warm-up	10 min	Discussion	Identify assets, attackers, entry points.
Authentication (AuthN)/authorization (AuthZ) walkthrough	18 min	Talk + code	Password hashing, JWT issuance, role-based access.
Micro demo: JWT decode	5 min	Live demo	<code>uv run python -m jwt decode</code> and explain claims.
Secrets hygiene & OWASP	12 min	Talk	.env discipline, least privilege, input validation, logging redaction.
<b>Part B – Lab 1</b>	<b>45 min</b>	<b>Guided coding</b>	<b>Add password hashing, login endpoint, JWT tokens.</b>
Break	10 min	—	Launch the shared <a href="#">10-minute timer</a> .
<b>Part C – Lab 2</b>	<b>45 min</b>	<b>Guided hardening</b>	<b>Role-based guards, security tests, secret scanning.</b>
Wrap-up	10 min	Questions and Answers (Q&A)	Confirm Exercise 3 (EX3) security tasks and remaining questions.

## Part A – Theory Highlights

1. **Threat modeling:** assets (user data, movie catalog), adversaries (students, bots), attack surface (login, recommendations, admin endpoints).
2. **Password storage:** never store plaintext; use `passlib.hash.bcrypt`. Clarify that a **salt** is a unique random string stored alongside each password hash to stop rainbow-table attacks, while a **pepper** is a separate secret kept outside the database so a stolen dump is still unusable. Close the loop with password policies students can adopt immediately.
3. **JSON Web Token (JWT) anatomy:** header, payload, signature; include `exp`, `iat`, `iss`, `aud`, `roles`. Keep tokens short-lived.
4. **Secrets hygiene:** `.env` vs `.env.example`, `docker compose` secrets, GitHub Actions secrets, never log tokens.
5. **OWASP highlights:** Broken auth, excessive data exposure, lack of rate limiting (already addressed). Tie back to previous sessions.



## Part B – Lab 1 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Wire bcrypt hashing and verify by logging a hashed password.
- **Minutes 10–25** – Implement `/token` endpoint, ensure fake user logins succeed.
- **Minutes 25–35** – Issue JWTs with roles/expiry and add settings entries.
- **Minutes 35–45** – Protect one endpoint with `require_role` and smoke-test with curl.

### 1. User model & hashing (`app/security.py`)

```

from datetime import datetime, timedelta
from typing import Optional

import jwt
from passlib.context import CryptContext

from app.config import Settings

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str) -> str:
 return pwd_context.hash(password)

def verify_password(plain_password: str, hashed: str) -> bool:
 return pwd_context.verify(plain_password, hashed)

def create_access_token(
 *,
 subject: str,
 settings: Settings,

```

```

 expires_delta: Optional[timedelta] = None,
 roles: Optional[list[str]] = None,
) -> str:
 now = datetime.utcnow()
 expire = now + (expires_delta or timedelta(minutes=30))
 payload = {
 "sub": subject,
 "iat": now.timestamp(),
 "exp": expire.timestamp(),
 "iss": settings.jwt_issuer,
 "aud": settings.jwt_audience,
 "roles": roles or ["student"],
 }
 return jwt.encode(payload, settings.jwt_secret, algorithm="HS256")

```

Add new settings (JWT secret, issuer, audience, token expiry). Store defaults in `.env.example` and populate `.env` locally.

Update `app/config.py`:

```

class Settings(BaseSettings):
 # existing fields...
 jwt_secret: str = "change-me"
 jwt_issuer: str = "movie-service"
 jwt_audience: str = "movie-clients"
 jwt_expiry_minutes: int = 30

```

## 2. Auth router (`app/routes/auth.py`)

```

from datetime import timedelta

from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm

from app.config import Settings
from app.security import create_access_token, verify_password

router = APIRouter(prefix="/token", tags=["auth"])

_FAKE_USERS = {
 "teacher": {
 "username": "teacher",
 "hashed_password": hash_password("classroom"),
 "roles": ["editor"],
 }
}

def authenticate(username: str, password: str):
 record = _FAKE_USERS.get(username)
 if not record or not verify_password(password, record["hashed_password"]):
 return None
 return record

@router.post("", response_model=dict)
def login(
 form: OAuth2PasswordRequestForm = Depends(),
 settings: Settings = Depends(),
) -> dict[str, str]:
 user = authenticate(form.username, form.password)
 if not user:

```

```

 raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid credentials")

 access_token = create_access_token(
 subject=user["username"],
 roles=user["roles"],
 settings=settings,
 expires_delta=timedelta(minutes=settings.jwt_expiry_minutes),
)
 return {"access_token": access_token, "token_type": "bearer"}

```

Mount router in `app/main.py` and guard protected endpoints.

 **Quick win:** When `uv run pytest tests/test_security.py::test_login_returns_token -q` returns a green dot, you've confirmed hashing + JWT issuance work together.

### 3. Dependency for protected routes

```

from fastapi import Depends, HTTPException, Security, status
from fastapi.security import OAuth2PasswordBearer

from app.config import Settings

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token")

def require_role(*allowed_roles: str):
 def _inner(token: str = Security(oauth2_scheme), settings: Settings = Depends()):
 try:
 payload = jwt.decode(
 token,
 settings.jwt_secret,
 algorithms=["HS256"],
 audience=settings.jwt_audience,
 issuer=settings.jwt_issuer,
)
 except jwt.PyJWTError as exc:
 raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token") from exc

 roles = set(payload.get("roles", []))
 if not roles.intersection(allowed_roles):
 raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Insufficient permissions")
 return payload

 return _inner

```

Apply `require_role("editor")` to sensitive routes (e.g., recommendation tools, admin updates).

## Part C – Lab 2 (45 Minutes)

### Lab timeline

- **Minutes 0–10** – Write security-focused tests for login and protected endpoints.
- **Minutes 10–25** – Add token expiry checks and role guard coverage.
- **Minutes 25–35** – Run secret scanners; update `docs/security-checklist.md`.
- **Minutes 35–45** – Demonstrate curl workflow (with/without token) and document remediation steps.

## 1. Security tests ( tests/test\_security.py )

```
from fastapi.testclient import TestClient

from app.main import app

client = TestClient(app)

def test_login_returns_token():
 response = client.post(
 "/token",
 data={"username": "teacher", "password": "classroom"},
 headers={"Content-Type": "application/x-www-form-urlencoded"},
)
 assert response.status_code == 200
 assert "access_token" in response.json()

def test_protected_endpoint_requires_token():
 response = client.post("/tool/recommend-movie", json={"payload": {"user_id": 1}})
 assert response.status_code == 401

def test_token_expiry(monkeypatch):
 settings = Settings(jwt_expiry_minutes=0)
 token = create_access_token(subject="user", settings=settings, expires_delta=timedelta(seconds=1))
 time.sleep(2)
 response = client.get("/movies", headers={"Authorization": f"Bearer {token}"})
 assert response.status_code in (401, 403)
```

(Remember to import missing modules in actual test file— `time` , `timedelta` , `Settings` , etc.)

## 2. Secret scanning

- Run `trufflehog filesystem --exclude .git .` or `gitleaks detect` and screenshot findings (should be clean).
- Add `make secrets-scan` target for CI.

## 3. OWASP checklist

Update `docs/security-checklist.md` with:

- Authentication enforced on admin endpoints.
- Input validation references (Pydantic validators from Sessions 03/05).
- Rate limiting (Session 10), logging redaction (avoid tokens).
- Secrets: `.env` variables, GitHub Action secrets, Docker secrets.

## 4. Optional stretch – refresh tokens

Create `/token/refresh` endpoint issuing new tokens if refresh token valid; store refresh tokens hashed in Redis.

### Verify tokens with curl (demo script)

```
Step 1: Fetch a token
TOKEN=$(curl -s -X POST http://localhost:8000/token \
```

```
-H "Content-Type: application/x-www-form-urlencoded" \
-d "username=teacher&password=classroom" \
| jq -r '.access_token')
```

```
Step 2: Call a protected endpoint
curl -H "Authorization: Bearer $TOKEN" \
http://localhost:8000/movies
```

```
Step 3: Confirm unauthorized request fails
curl -i http://localhost:8000/movies
```

Encourage students to add the script to `scripts/security-smoke.sh` so QA teams can reuse it.

## Wrap-up & Next Steps

-  Password hashing, JWT issuance, role guards, security tests, secret scanning.
- Finalize documentation, tool-friendly API design, publish `docs/service-contract.md` updates, and gather all EX3 deliverables.

## Troubleshooting

- **bcrypt missing build prerequisites** → ensure `rust / gcc` installed or use prebuilt wheels (should work on macOS/Linux). On Windows/WSL, install `build-essential`.
- **Invalid JWT** → verify `aud / iss` match; align Settings across services (worker, API, CLI).
- **Token not accepted by Swagger UI** → set OAuth2 config in FastAPI app: `app = FastAPI(swagger_ui_init_oauth={"clientId": "web"})` if needed.

## Common pitfalls

- **Clock skew causing “token expired” errors** – sync host clocks or inject fixed datetimes during tests (e.g., via `freezegun`).
- **Secrets checked into git** – run `trufflehog` before commits and rotate anything flagged immediately.
- **Role mismatch** – log decoded payload roles locally (strip in production) and ensure `_FAKE_USERS` entries match `require_role` guards.

## Student Success Criteria

By the end of Session 11, every student should be able to:

- ☐ Hash and verify passwords with `passlib[bcrypt]` instead of storing plaintext credentials.
- ☐ Issue and validate JWTs (with `iss`, `aud`, `exp`, and roles) that gate at least one endpoint.
- ☐ Run security smoke tests (pytest + curl) and share clean secret-scan results.

**If a student cannot check a box, schedule a security-focused pairing soon after this session.**

## AI Prompt Seeds

- “Generate FastAPI login endpoint that hashes passwords with passlib, returns JWT with exp/iss/aud claims.”
- “Write pytest tests for 401/403 scenarios against a JWT-protected endpoint.”
- “Create a security checklist covering OWASP API Top 3 for a FastAPI service.”

# Session 12 – Tool-Friendly APIs & FastMCP Finish Line

docs/sessions/session-12.md

- **Date:** Monday, 01/06/2026
- **Theme:** Lock down the EX3 release by polishing every externally facing surface—pagination, ETags, docs, automation—and show how the same contracts plug into FastMCP so tools can call your API hands-free.

## Session Story

You enter Session 12 with a production-shaped stack: FastAPI + Postgres + Redis + background workers + Streamlit/React surfaces and security controls. Now it is about finish-line polish—make responses deterministic, document them rigorously, and prove automation + tooling are airtight so EX3 demos impress external reviewers. This meeting is a release-engineering sprint: align the REST contract with CSV + pagination, add caching clues, run every lint/test/doc tool, and mirror the contract via FastMCP so assistants can call the service without custom glue.

## Learning Objectives

- Publish deterministic REST responses with pagination, CSV export, and OpenAPI examples that mirror your clients.
- Return caching-friendly metadata (ETag + `If-None-Match`) and document the contract inside `docs/service-contract.md`.
- Automate quality gates (MkDocs/pdocs, Ruff, mypy, pre-commit, changelog) so releases are repeatable.
- Bridge the API to FastMCP by exposing at least one MCP tool that reuses the `/movies` vocabulary.

## What You'll Build

- Pagination helpers + query dependencies ( `page` , `page_size` , `format` ) and updated `/movies` FastAPI route that returns metadata + deterministic ordering.
- CSV export path powered by `csv.DictWriter` (or pandas) and `StreamingResponse`.
- ETag middleware/utility that hashes response payloads and honors `If-None-Match` for 304 responses.
- `docs/service-contract.md` + OpenAPI examples that describe pagination headers, CSV shape, and caching semantics.
- Release automation: MkDocs/pdocs build, Ruff, mypy, pytest, Schemathesis, `pre-commit`, and `docs/release-checklist.md`.
- A FastMCP tool ( `scripts/movies_mcp.py` + `scripts/mcp_probe.py` ) that reuses the same repository calls as REST.

## Prerequisites & JiTT

1. Pull latest EX3 changes and make sure `docker compose up --build`, `uv run pytest --cov`, and `uv run schemathesis run docs/contracts/openapi.json --dry-run` all succeed.
2. Install finishing tools if missing:

```
uv add mkdocs-material pdocs ruff mypy pre-commit
uv add "mcp[cli]>=1.1.0"
```

- Seed at least six movies ( `uv run python scripts/db.py bootstrap --sample 6` ) so pagination and CSV exports show interesting data.
- Create/refresh `docs/release-checklist.md` with owner, trigger commands, screenshot/log requirements, and FastMCP verification steps.
- Sync `.env` + `.env.example` with any new settings ( `MOVIE_LIST_DEFAULT_PAGE_SIZE` , `MOVIE_ETAG_SALT` , etc.).

🔗 **EX3 Deliverable:** Pagination + ETag on one list endpoint, CSV export, OpenAPI examples, MkDocs/pdocs + Ruff + mypy + pre-commit automation, a FastMCP probe that replays the same contracts, and an updated release checklist.

## Toolkit Snapshot

- **FastAPI + SQLAlchemy** – still the backbone; today you add pagination params, CSV routes, and deterministic ordering.
- **httpx + pandas/csv module** – verifying CSV output and writing quick smoke scripts.
- **uv** – runs lint/test/doc commands and the FastMCP probe.
- **Ruff + mypy + pytest + Schemathesis** – quality gates required before you tag EX3.
- **MkDocs/pdocs** – generate docs directly from code, linking OpenAPI + service contract references.
- **pre-commit** – enforces formatting, Ruff, mypy, doc builds before pushing.
- **FastMCP** – exposes the same movie vocabulary through Model Context Protocol for assistants/tools.

## Agenda

Segment	Duration	Format	Focus
EX3 dry run	10 min	Demos	Walk through local run + contracts.
Tool-friendly patterns	15 min	Talk	Pagination, deterministic responses, CSV notes.
FastMCP spotlight	10 min	Live demo	One REST verb mirrored as an MCP tool.
Release hygiene	10 min	Talk	Docs, lint, tests, changelog.
<b>Part B – Lab 1 (Polish)</b>	<b>45 min</b>	<b>Guided build</b>	Pagination, ETag, CSV export, OpenAPI examples.
<b>Part C – Lab 2 (Automation)</b>	<b>45 min</b>	<b>Guided runbook</b>	Docs build, <code>pre-commit run --all-files</code> , release checklist rehearsal, FastMCP probe.
Closing circle	5 min	Discussion	Commitments for EX3 showcase.

## Part A – Core Moves

1. **Deterministic responses:** Add `page` , `page_size` , `sort` defaults, `X-Total-Count` , `X-Total-Pages` , and CSV output so every client/tool gets predictable slices.
2. **Cache hints:** Compute an ETag (hash of response payload + salt) and honor `If-None-Match` to return `304 Not Modified` when the requested page hasn't changed.
3. **Contract-first docs:** Copy the example response + CSV header into OpenAPI `examples` and `docs/service-contract.md` .
4. **Automation:** Wrap Ruff, mypy, pytest, Schemathesis, MkDocs, and FastMCP probe inside your release checklist + CI.

5. **FastMCP bridge:** Mirror nouns/verbs from REST when building MCP tools so AI assistants can reason about the same schema.



## Part B – Lab 1: Deterministic list experience (45 minutes)

### Step 0 – Confirm data + API

Keep `uv run uvicorn movie_service.app.main:app --reload` running. Hit `curl http://localhost:8000/movies | jq length` to prove multiple rows exist.

### Step 1 – Pagination helper + query dependency

`movie_service/app/pagination.py`:

```
from dataclasses import dataclass
from fastapi import Query

@dataclass
class Page:
 number: int
 size: int

def pagination_params(
 page: int = Query(1, ge=1),
 page_size: int = Query(5, ge=1, le=100),
) -> Page:
 return Page(number=page, size=page_size)
```

Import `pagination_params` inside your movies router and wire it as a dependency.

### Step 2 – Update GET /movies

```
@router.get("/movies", response_model=MoviePagedResponse)
def list_movies(
 pagination: Page = Depends(pagination_params),
 format: Literal["json", "csv"] = Query("json"),
 repo: MovieRepository = Depends(get_repository),
) -> Response:
 movies, total = repo.list_paginated(pagination.number, pagination.size)
 payload = {
 "page": pagination.number,
 "page_size": pagination.size,
 "total": total,
 "items": movies,
 }
 if format == "json":
 return JSONResponse(payload, headers={"X-Total-Count": str(total)})
 return stream_as_csv(payload)
```

Add deterministic ordering (e.g., `ORDER BY created_at, id`) inside `repository_db.py`.

### Step 3 – CSV streaming helper

```
import csv
from io import StringIO
from fastapi.responses import StreamingResponse

def stream_as_csv(payload: dict[str, object]) -> StreamingResponse:
 buffer = StringIO()
 fieldnames = ("id", "title", "year", "genre")
 writer = csv.DictWriter(buffer, fieldnames=fieldnames)
 writer.writeheader()
 for row in payload["items"]:
 writer.writerow(row)
 buffer.seek(0)
 headers = {
 "Content-Type": "text/csv",
 "Content-Disposition": 'attachment; filename="movies.csv"',
 "X-Total-Count": str(payload["total"]),
 }
 return StreamingResponse(iter([buffer.read()]), headers=headers)
```

Smoke test:

```
curl "http://localhost:8000/movies?page=1&page_size=3&format=csv"
```

### Step 4 – ETag utility

movie\_service/app/etag.py:

```
import hashlib
import json
from fastapi import Request, Response

def compute_etag(body: dict) -> str:
 digest = hashlib.sha256(json.dumps(body, sort_keys=True).encode("utf-8")).hexdigest()
 return f'W/"{digest}"'

def maybe_return_not_modified(request: Request, response: Response, payload: dict) -> Response:
 etag = compute_etag(payload)
 if_none_match = request.headers.get("if-none-match")
 if if_none_match == etag:
 not_modified = Response(status_code=304)
 not_modified.headers["ETag"] = etag
 return not_modified
 response.headers["ETag"] = etag
 return response
```

Wrap the JSON/CSV responses:

```
response = JSONResponse(payload, headers={"X-Total-Count": str(total)})
return maybe_return_not_modified(request, response, payload)
```

Verify loop:

```
curl -i "http://localhost:8000/movies?page=1&page_size=3" | tee /tmp/page1.out
curl -i "http://localhost:8000/movies?page=1&page_size=3" -H 'If-None-Match: W/"...hash..."'
```

## Step 5 – Document everything

Update `docs/service-contract.md` and OpenAPI `response_model` examples with:

- JSON payload (page/page\_size/total/items)
- CSV header order, quoting rules
- Header expectations ( `X-Total-Count` , `ETag` , `If-None-Match` )
- Deterministic sorting guarantee + acceptable page sizes

## Part C – Lab 2: Automation, docs, and FastMCP (45 minutes)

---

### Step 1 – One-command release check

Create `scripts/check_release.py` (Typer or plain Python) that wraps the commands you expect before tagging:

```
uv run ruff check .
uv run mypy movie_service
uv run pytest --cov=movie_service --cov-report=term-missing
uv run schemathesis run http://localhost:8000/openapi.json --checks all
uv run mkdocs build
uv run pdocs movie_service.app.main
```

Log success/failure in the release checklist.

### Step 2 – pre-commit hooks

`.pre-commit-config.yaml` snippet:

```
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
 rev: v0.5.5
 hooks:
 - id: ruff
 - id: ruff-format
- repo: https://github.com/pre-commit/mirrors-mypy
 rev: v1.11.0
 hooks:
 - id: mypy
 additional_dependencies: ["sqlmodel", "pydantic"]
- repo: local
 hooks:
 - id: mkdocs
 name: mkdocs build
 entry: uv run mkdocs build
 language: system
 pass_filenames: false
```

Run `pre-commit install` and `pre-commit run --all-files`.

### Step 3 – Release checklist + docs

`docs/release-checklist.md` should include:

- Owner + timestamp of dry run.

- Evidence links (screenshots, `logs/release-2026-06-01.txt`).
- Commands run + exit codes.
- FastMCP probe instructions + expected output snippet.

Update `docs/index.md` / `docs/service-contract.md` to link to release checklist and CSV/ETag sections.

## Step 4 – FastMCP lightning lab

`scripts/movies_mcp.py`:

```
from mcp.server.fastmcp import FastMCP
from movie_service.app.repositories import MovieRepository, get_session

mcp = FastMCP("io.eass.movies")

@mcp.tool(name="list-movies-page")
async def list_movies_page(page: int = 1, page_size: int = 5) -> dict:
 repo = MovieRepository(get_session())
 items, total = repo.list_paginated(page, page_size)
 return {"status": 200, "total": total, "items": items}

if __name__ == "__main__":
 mcp.run(transport="stdio")
```

Probe with:

```
uv run python scripts/mcp_probe.py --tool list-movies-page --page 1 --page-size 3
```

Copy output into the release checklist/log so graders can see evidence.

## Step 5 – Tag-prep retrospective

- Capture final TODOs in `docs/EX3-notes.md` (telemetry screenshots, FastMCP output, CSV samples).
- Ensure `CHANGELOG.md` or `README` includes Session 12 highlights (deterministic API, FastMCP bridge).

## Simple Demo – Pagination + ETag handshake

1. Boot the API (local run or Compose) and seed at least 6 movies so pagination is visible.
2. Request the first page and capture the ETag:  
`curl -i "http://localhost:8000/movies?page=1&page_size=3" | tee /tmp/movies.out`
3. Reuse the tag: `curl -i "http://localhost:8000/movies?page=1&page_size=3" -H "If-None-Match: <etag-from-step-2>"` and confirm FastAPI replies `304 Not Modified`; repeat with `?format=csv` to verify the export.

## Wrap-up & Next Steps

- ☒ Pagination metadata, CSV exports, `If-None-Match` handling, and updated service contract.
- ☒ Release automation documented and rehearsed ( `ruff` , `mypy` , `pytest` , `Schemathesis` , `MkDocs` , FastMCP probe, pre-commit).
- ☒ FastMCP tool mirrors `/movies` , giving assistants parity with REST clients.
- Action items: freeze EX3 feature work, focus on polish + rehearsals, and capture demo evidence (screenshots, trace IDs, release logs) before showcase day.

## Troubleshooting

---

- **304 never returned:** Ensure `If-None-Match` header matches the exact `ETag` value (quotes included) and that payload ordering stays deterministic.
- **CSV output garbled:** Double-check newline handling on Windows ( `newline=""` when creating `StringIO` ) and verify numeric fields aren't quoted as strings if your clients expect ints.
- **MkDocs build fails:** Confirm `mkdocs.yml` references existing docs; add pagination/CSV topics to the nav tree.
- **pre-commit too slow:** Use `SKIP=mkdocs pre-commit run --all-files` when iterating locally, but run full hooks before merging.
- **FastMCP tool import errors:** Run from repo root with `PYTHONPATH=.` or convert scripts into modules under `scripts/` to avoid relative import pain.

## Student Success Criteria

---

- ☐ `GET /movies` returns deterministic JSON with pagination metadata and `ETag` headers, plus optional CSV export.
- ☐ `docs/service-contract.md`, OpenAPI examples, and release checklist describe pagination/CSV/ETag behavior.
- ☐ `pre-commit run --all-files`, `uv run mkdocs build`, `uv run ruff check`, `uv run mypy`, `uv run pytest`, and `uv run schemathesis` all pass locally.
- ☐ FastMCP probe executes successfully and is documented with usage instructions + sample output.

Schedule a mentor pairing if any box remains unchecked; EX3 grading hinges on this polish.

## AI Prompt Seeds

---

- “Given this FastAPI pagination spec, generate the route + repository code (with deterministic ordering, CSV export, ETag support) and corresponding tests.”
- “Draft a release checklist for FastAPI including Ruff, mypy, pytest, Schemathesis, MkDocs, and FastMCP probes.”
- “Create a FastMCP tool that mirrors `/movies` pagination and explain how to document the tool in `docs/service-contract.md`.”

# Optional DuckDB Mini-Lakehouse Lab

docs/sessions/optional/DuckDBMiniLakehouse.md

This optional lab bolts onto **Session 05 – PostgreSQL Foundations** (after students have seen SQLite in Session 04) or can run as a standalone 2×45-minute workshop. The goal is to help the cohort feel how DuckDB behaves as “SQLite for analytics” with a mini medallion layout and a word-count style map/reduce query—using nothing beyond their laptop. Pair this with the CodeCut deep dive (<https://codecut.ai/deep-dive-into-duckdb-data-scientists/>) so students understand why DuckDB’s columnar, vectorized core excels at analytical workloads.

Learners finish with:

- 1. A reproducible directory structure for Bronze → Silver → Gold Parquet data.
- 2. Two DuckDB files (PROD + DEV) wired together with environment variables.
- 3. CLI and Python patterns for exploring data, refreshing DEV from PROD, and creating views.
- 4. A “map/reduce” style aggregation that mimics word count across a sample event log.

## Before Lab – Preflight (JiTT)

- Install DuckDB CLI (macOS: `brew install duckdb`, Linux: download the static binary from [duckdb.org](https://duckdb.org), Windows: use winget/MSI).
- Ensure `uv` is on PATH (already required for the course).
- Clone a scratch repo or reuse the EX3 workspace—no external services required.
- Optional: `uv add duckdb polars pandas` if students prefer notebook/Python exploration.

## Agenda (2×45 Minutes)

Segment	Duration	Format	Focus
Medallion primer + folder scaffold	20 min	Talk + shell	Bronze/Silver/Gold directories, DEV vs PROD databases.
DuckDB CLI tour	25 min	Live demo	Environment variables, SQL scripts, attaching prod to dev.
Break	10 min	—	Encourage stretch + hydration.
Map/Reduce walk-through	20 min	Guided coding	Word-count style aggregation over parquet event logs.
Python automation	25 min	Guided coding	One Typer script to refresh DEV + run analytics.
Wrap	10 min	Q&A	How to tie this into EX3 without expanding grade scope.

## Part A – Scaffold the Mini Lakehouse (45 Minutes)

### 0. Verify the CLI is ready

```
duckdb --version
```

If the command prints a version (e.g., `DuckDB 0.10.2`), you are set. Otherwise, install DuckDB before proceeding.

## 1. Directory structure + environment variables

From the repo root (or a scratch directory), create the folders and export variables:

```
export DUCK_WH_DB="$PWD/duck/warehouse/sports_ml_warehouse.duckdb"
export DUCK_DEV_DB="$PWD/duck/dev/sports_ml_warehouse.duckdb"
export LAKE_ROOT="$PWD/duck/lake"

mkdir -p "$(dirname "$DUCK_WH_DB")" "$(dirname "$DUCK_DEV_DB")"/backups \
"$LAKE_ROOT"/{bronze,silver,gold}
```

Tip: Drop the three `export` commands into `scripts/duckdb.env` so students can `source scripts/duckdb.env` later:

```
mkdir -p scripts
cat <<'EOF' > scripts/duckdb.env
export DUCK_WH_DB="$PWD/duck/warehouse/sports_ml_warehouse.duckdb"
export DUCK_DEV_DB="$PWD/duck/dev/sports_ml_warehouse.duckdb"
export LAKE_ROOT="$PWD/duck/lake"
EOF
```

Afterwards, activate the variables in any new shell:

```
source scripts/duckdb.env
```

## 2. Seed Bronze data with Python (synthetic but realistic)

```
uv run python - <<'PY'
from pathlib import Path
import random
import json
import os

lake_root = Path(os.environ["LAKE_ROOT"])
root = lake_root / "bronze" / "events"
root.mkdir(parents=True, exist_ok=True)

actors = ["student", "instructor", "assistant"]
verbs = ["viewed", "updated", "deleted", "created"]
entities = ["movie", "rating", "playlist", "profile"]

records = []
random.seed(42)
for event_id in range(1, 10_001):
 records.append(
 {
 "event_id": event_id,
 "actor": random.choice(actors),
 "verb": random.choice(verbs),
 "entity": random.choice(entities),
 "duration_ms": random.randint(50, 2_500),
 }
)

bronze_file = root / "events.jsonl"
bronze_file.write_text("\n".join(json.dumps(r) for r in records))
print(f"Wrote {bronze_file} with {len(records)} events")
PY
```

### 3. Load Bronze → Silver (cleaned parquet) via DuckDB CLI

```
duckdb "$DUCK_WH_DB" -c "
 CREATE SCHEMA IF NOT EXISTS bronze;
 CREATE SCHEMA IF NOT EXISTS silver;

 COPY (
 SELECT
 event_id,
 lower(actor) AS actor,
 lower(verb) AS verb,
 lower(entity) AS entity,
 duration_ms
 FROM read_json('$LAKE_ROOT/bronze/events/events.jsonl')
)
 TO '$LAKE_ROOT/silver/events/events.parquet'
 (FORMAT 'parquet', COMPRESSION 'zstd');
"
```

### 4. Build PROD warehouse references

```
duckdb "$DUCK_WH_DB" -c "
 CREATE OR REPLACE TABLE bronze.events AS
 SELECT * FROM read_json('$LAKE_ROOT/bronze/events/events.jsonl');

 CREATE OR REPLACE TABLE silver.events AS
 SELECT * FROM read_parquet('$LAKE_ROOT/silver/events/events.parquet');

 CREATE VIEW IF NOT EXISTS gold.event_summary AS
 SELECT
 actor,
 verb,
 COUNT(*) AS event_count,
 AVG(duration_ms)::INTEGER AS avg_duration_ms
 FROM silver.events
 GROUP BY actor, verb
 ORDER BY event_count DESC;
"
```

At this point, students have a persistent `.duckdb` file with Bronze (raw JSON view), Silver (clean parquet), and a Gold aggregate view.

---

## Part B – Map/Reduce Style Analytics (45 Minutes)

### 1. Refresh DEV from PROD without copying raw files

```
duckdb "$DUCK_DEV_DB" -c "
 ATTACH '$DUCK_WH_DB' AS prod (READ_ONLY);

 CREATE SCHEMA IF NOT EXISTS bronze_views;
 CREATE OR REPLACE VIEW bronze_views.events AS
 SELECT * FROM prod.bronze.events;

 CREATE SCHEMA IF NOT EXISTS gold_labs;
 CREATE OR REPLACE TABLE gold_labs.event_word_counts AS
 SELECT lower(token) AS token, COUNT(*) AS hits
 FROM prod.silver.events,
```

```

 UNNEST(string_split(verb || ' ' || entity, ' ')) AS token
 GROUP BY lower(token)
 ORDER BY hits DESC;
"

```

The `UNNEST(string_split(...))` pattern behaves like a classic map/reduce word count: **map** splits each verb/entity pair into tokens, **reduce** aggregates counts. Students can compare with the canonical Hadoop example, but here it runs locally in milliseconds.

Verify the Gold tables exist and contain data:

```

duckdb "$DUCK_DEV_DB" -c "
 SHOW TABLES;
"

```

## 2. Explore the results (CLI)

```

duckdb "$DUCK_DEV_DB" -c "
 SELECT * FROM gold_labs.event_word_counts LIMIT 10;
"

```

## 3. Bonus: Sliding window aggregations

Demonstrate richer analytics without leaving DuckDB:

```

duckdb "$DUCK_DEV_DB" -c "
 ATTACH '$DUCK_WH_DB' AS prod (READ_ONLY);
 SELECT
 verb,
 COUNT(*) AS total_events,
 AVG(duration_ms) AS avg_duration,
 PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY duration_ms) AS p90_duration
 FROM prod.silver.events
 GROUP BY verb
 ORDER BY total_events DESC;
"

```

## 4. Python helper to automate refresh + query (Typer CLI)

```

uv add typer duckdb
cat <<'PY' > duck_tools.py
import duckdb
import typer
import os

app = typer.Typer(help="DuckDB mini-lakehouse helpers")

def _require_env(name: str) -> str:
 value = os.getenv(name)
 if not value:
 raise typer.BadParameter(f"Environment variable {name} is not set")
 return value

@app.command()
def refresh_dev() -> None:
 dev = _require_env("DUCK_DEV_DB")
 prod = _require_env("DUCK_WH_DB")

```

```

with duckdb.connect(dev) as con:
 con.execute("ATTACH ? AS prod (READ_ONLY)", [prod])
 con.execute(
 """
 CREATE OR REPLACE TABLE gold_labs.event_word_counts AS
 SELECT lower(token) AS token, COUNT(*) AS hits
 FROM prod.silver.events,
 UNNEST(string_split(verb || ' ' || entity, ' ')) AS token
 GROUP BY lower(token)
 ORDER BY hits DESC
 """
)
 typer.echo("Refreshed DEV gold_labs.event_word_counts")

@app.command()
def top_tokens(limit: int = 10) -> None:
 dev = _require_env("DUCK_DEV_DB")
 with duckdb.connect(dev) as con:
 rows = con.execute(
 "SELECT token, hits FROM gold_labs.event_word_counts LIMIT ?", [limit]
).fetchall()
 for token, hits in rows:
 typer.echo(f"{token:>10} : {hits}")

if __name__ == '__main__':
 app()
PY

```

Usage:

```

source scripts/duckdb.env
uv run python duck_tools.py refresh-dev
uv run python duck_tools.py top-tokens --limit 5

```

## Troubleshooting Cheatsheet

- **“Conflicting lock is held”**: ensure no other DuckDB session is open ( `ps aux | rg duckdb` ), then remove stale `*.duckdb.wal` files if present.
- **“No such file or directory” errors**: confirm the three environment variables ( `DUCK_WH_DB` , `DUCK_DEV_DB` , `LAKE_ROOT` ) point to valid paths; rerun the `mkdir -p ...` command.
- **Queries return zero rows**: re-run the Bronze → Silver load step to regenerate Parquet, then refresh views.
- **Typer CLI fails**: verify `uv add typer duckdb` succeeded and that you are sourcing the env file before running commands.

## Quick Demo – From Empty Folder to Insights

Paste each block (after sourcing `scripts/duckdb.env` ) to go from nothing to a working DuckDB analytics sandbox:

```

1. Scaffold folders and environment
mkdir -p duck && mkdir -p scripts
cat <<'EOF' > scripts/duckdb.env
export DUCK_WH_DB="$PWD/duck/warehouse/sports_ml_warehouse.duckdb"
export DUCK_DEV_DB="$PWD/duck/dev/sports_ml_warehouse.duckdb"

```

```

export LAKE_ROOT="$PWD/duck/lake"
EOF
source scripts/duckdb.env
mkdir -p "${dirname "$DUCK_WH_DB"}" "${dirname "$DUCK_DEV_DB"}"/backups \
 "$LAKE_ROOT"/{bronze,silver,gold}

2. Generate Bronze JSON
uv run python - <<'PY'
from pathlib import Path
import random
import json
import os

lake_root = Path(os.environ["LAKE_ROOT"])
root = lake_root / "bronze" / "events"
root.mkdir(parents=True, exist_ok=True)

actors = ["student", "instructor", "assistant"]
verbs = ["viewed", "updated", "deleted", "created"]
entities = ["movie", "rating", "playlist", "profile"]

records = []
random.seed(42)
for event_id in range(1, 10_001):
 records.append(
 {
 "event_id": event_id,
 "actor": random.choice(actors),
 "verb": random.choice(verbs),
 "entity": random.choice(entities),
 "duration_ms": random.randint(50, 2_500),
 }
)

bronze_file = root / "events.jsonl"
bronze_file.write_text("\n".join(json.dumps(r) for r in records))
print(f"Wrote {bronze_file} with {len(records)} events")
PY

3. Load into PROD and build Gold view
duckdb "$DUCK_WH_DB" -c "CREATE SCHEMA IF NOT EXISTS bronze; CREATE SCHEMA IF NOT EXISTS silver;"
duckdb "$DUCK_WH_DB" -c "COPY (SELECT event_id, lower(actor) AS actor, lower(verb) AS verb,
lower(entity) AS entity, duration_ms FROM read_json('$LAKE_ROOT/bronze/events/events.jsonl')) TO
'$LAKE_ROOT/silver/events/events.parquet' (FORMAT 'parquet', COMPRESSION 'zstd');"
duckdb "$DUCK_WH_DB" -c "CREATE OR REPLACE TABLE bronze.events AS SELECT * FROM
read_json('$LAKE_ROOT/bronze/events/events.jsonl'); CREATE OR REPLACE TABLE silver.events AS SELECT *
FROM read_parquet('$LAKE_ROOT/silver/events/events.parquet'); CREATE VIEW IF NOT EXISTS
gold.event_summary AS SELECT actor, verb, COUNT(*) AS event_count, AVG(duration_ms)::INTEGER AS
avg_duration_ms FROM silver.events GROUP BY actor, verb ORDER BY event_count DESC;"

4. Refresh DEV views and run map/reduce
duckdb "$DUCK_DEV_DB" -c "ATTACH '$DUCK_WH_DB' AS prod (READ_ONLY); CREATE SCHEMA IF NOT EXISTS
bronze_views; CREATE OR REPLACE VIEW bronze_views.events AS SELECT * FROM prod.bronze.events; CREATE
SCHEMA IF NOT EXISTS gold_labs; CREATE OR REPLACE TABLE gold_labs.event_word_counts AS SELECT
lower(token) AS token, COUNT(*) AS hits FROM prod.silver.events, UNNEST(string_split(verb || ' ' ||
entity, ' ')) AS token GROUP BY lower(token) ORDER BY hits DESC;"
duckdb "$DUCK_DEV_DB" -c "SELECT * FROM gold_labs.event_word_counts LIMIT 5;"

```

Replace the `uv run python` line with the inline script from Part A, Step 2 if you prefer not to store a separate helper file.

## Where This Fits in the Course

---

- Use it after **Session 05** to contrast transactional SQLite (row-oriented) with analytical DuckDB (columnar).
- Reinforce local-first analytics: everything runs without cloud accounts, matching the course ethos.
- Treat the map/reduce demo as a bridge to big-data concepts without leaving a single laptop.
- Optional extension: challenge advanced students to add a Gold table that rolls up activity per 5-minute interval using `WINDOW` functions.

# Team Topologies: Organizing Business and Technology Teams for Fast Flow

---

docs/sessions/optional/TeamTopologies.md

## Key Concepts

---

### 1. Four Fundamental Team Types

- i. Stream-aligned Teams • Deliver end-to-end value aligned to a workflow, product, or user journey. • Examples: Product development teams, feature teams.
- ii. Enabling Teams • Support other teams in adopting new tools, techniques, or skills. • Act as mentors or consultants to unblock and empower teams.
- iii. Complicated Subsystem Teams • Handle specialized, complex parts of a system requiring niche expertise. • Examples: Advanced algorithms, machine learning frameworks.
- iv. Platform Teams • Provide internal tools, infrastructure, or APIs to enable self-service. • Examples: DevOps tools, cloud infrastructure, CI/CD pipelines.

### 2. Three Interaction Modes

- i. Collaboration • Short-term partnerships for problem-solving or innovation.
- ii. X-as-a-Service • Teams consume services or APIs from other teams.
- iii. Facilitating • Enabling teams mentor or coach other teams for skill development or problem resolution.

### 3. Cognitive Load Management • Each team should handle a manageable amount of cognitive load. • Teams should be designed to focus on their specific responsibilities without being overwhelmed.

### 4. Fast Flow Principles • Optimize team structures to deliver value quickly. • Minimize dependencies and ensure clear boundaries between teams.

### 5. Conway's Law • Team structures should reflect the desired software architecture to avoid mismatches.

Why It Matters • Adapts organizations to modern software development practices like DevOps and Agile. • Reduces friction, boosts team autonomy, and improves productivity. • Ensures sustainable delivery by focusing on team capabilities and flow.

Key Takeaways • Choose the right team type for the right job (Stream-aligned, Enabling, Complicated Subsystem, Platform). • Align interactions to minimize friction (Collaboration, X-as-a-Service, Facilitating). • Design for fast flow by reducing cognitive load and aligning team structures with architecture.

Actionable Insights • Streamline team responsibilities to avoid bottlenecks. • Foster autonomy through clear boundaries and reusable platforms. • Use enabling teams to unblock and upskill others without creating dependencies.

# Optional MCP Workshop – Weather MCP Server

docs/sessions/optional/mcp.md

This add-on session slots in after **Session 12 – Tool-Friendly APIs** (or during the EX3 milestone sprint) to show how to publish Model Context Protocol tools end-to-end. It reuses the same 3×45-minute cadence as the core sessions and keeps the rest of the syllabus untouched.

- **Instructor GitHub handle:** `zozo123`
- **Docker Hub namespace:** `zozo001`

Learners finish with:

1. A FastMCP server exposing two tools backed by real weather data.
2. A FastAPI wrapper that mounts Streamable HTTP at `/mcp`.
3. Clients for stdio and HTTP transports.
4. A Docker image pushed to Docker Hub ( `zozo001/weather-mcp` ).
5. A validated `server.json` published to the official MCP Registry via OCI.

## Before Workshop – Preflight (JiTT)

- Install dependencies ahead of time:

```
uv add mcp[cli] fastapi uvicorn httpx logfire
```
- Make sure Docker is logged in to `zozo001` and `uv run python -m mcp --help` succeeds.
- Review the Logfire FastMCP guide (link in LMS) and note how to toggle telemetry on/off ( `LOGFIRE_SEND_OFFLINE=true|false` ).
- Export NOAA API policy reminder: include `User-Agent` header with contact info.

## Agenda (3×45 Minutes)

Segment	Duration	Format	Focus
MCP primer + scaffold	45 min	Talk + live coding	Architecture, repo layout, FastMCP tools
Local/remote runs	45 min	Guided coding	stdio + HTTP servers, Inspector + Python clients
Shipping & registry	45 min	Guided coding	Docker build/push, server.json, MCP Registry publish

## Learning Objectives

- Explain MCP transports (stdio vs Streamable HTTP) and when to use each.
- Scaffold a FastMCP server with deterministic tool schemas.
- Run and validate the server locally, then containerize and push it.
- Publish the server to the MCP Registry using OCI images from Docker Hub.

## Part A – Primer & Project Scaffold (45 min)

### 1. Architecture slide

Claude Desktop (stdio) / Claude Connector (HTTP) → MCP server → tools → upstream APIs. Emphasize deterministic schemas, no stdout noise, and reuse of `uv` workflows.

### 2. Create repo scaffold

```
git clone https://github.com/zozo123/weather-mcp.git
cd weather-mcp
uv init --python 3.12
uv add "mcp[cli]" httpx fastapi uvicorn
```

Repo layout:

```
weather-mcp/
├── pyproject.toml
├── weather_mcp.py
├── app.py
├── Dockerfile
├── .dockerignore
├── client_stdio.py
├── client_http.py
└── README.md (optional)
```

### 3. `pyproject.toml`

```
[project]
name = "weather-mcp"
version = "0.1.0"
description = "Weather MCP demo with alerts and forecast tools"
requires-python = ">=3.12"
dependencies = [
 "mcp[cli]",
 "httpx>=0.27.0",
 "fastapi>=0.115.0",
 "uvicorn>=0.30.0",
]

[tool.uv]
```

### 4. `weather_mcp.py`

```
from typing import Any
import httpx
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("io.docker.zozo001.weather-mcp")

NWS_API_BASE = "https://api.weather.gov"
USER_AGENT = "weather-mcp/0.1.0"

async def _get_json(url: str) -> dict[str, Any] | None:
 headers = {"User-Agent": USER_AGENT, "Accept": "application/geo+json"}
 async with httpx.AsyncClient() as client:
```

```

 try:
 resp = await client.get(url, headers=headers, timeout=30.0)
 resp.raise_for_status()
 return resp.json()
 except Exception:
 return None

def _format_alert(feature: dict) -> str:
 props = feature.get("properties", {})
 return (
 f"Event: {props.get('event', 'Unknown')}\n"
 f"Area: {props.get('areaDesc', 'Unknown')}\n"
 f"Severity: {props.get('severity', 'Unknown')}\n"
 f>Description: {props.get('description', 'No description')}\n"
 f"Instructions: {props.get('instruction', 'None')}"
)

@mcp.tool()
async def get_alerts(state: str) -> str:
 """Get active weather alerts for a US state (e.g., 'CA')."""
 data = await _get_json(f"{NWS_API_BASE}/alerts/active/area/{state}")
 if not data or "features" not in data:
 return "No alerts or fetch failed."
 items = data.get("features", [])
 if not items:
 return "No active alerts."
 return "\n---\n".join(_format_alert(f) for f in items)

@mcp.tool()
async def get_forecast(latitude: float, longitude: float) -> str:
 """Get a short forecast for a US lat/long."""
 points_data = await _get_json(f"{NWS_API_BASE}/points/{latitude},{longitude}")
 if not points_data:
 return "Failed to resolve forecast grid."
 forecast_data = await _get_json(points_data["properties"]["forecast"])
 if not forecast_data:
 return "Detailed forecast unavailable."
 periods = forecast_data["properties"]["periods"][:5]
 sections = []
 for p in periods:
 sections.append(
 f"{p['name']}: {p['temperature']}{p['temperatureUnit']}, "
 f"Wind {p['windSpeed']} {p['windDirection']}\n{p['detailedForecast']}"
)
 return "\n---\n".join(sections)

if __name__ == "__main__":
 mcp.run(transport="stdio")

```

## 4a. Optional telemetry toggle (Logfire)

Add `telemetry.py`:

```

import os

import logfire

def configure() -> None:
 enabled = os.getenv("LOGFIRE_ENABLED", "false").lower() == "true"
 logfire.configure(
 service_name="weather-mcp",

```

```
 send_to_logfire=enabled,
)
```

Call `telemetry.configure()` near the top of `weather_mcp.py`. This mirrors the observability discipline from Session 07 and allows instructors to switch telemetry on/off via environment variable.\n+

## 5. `app.py`

```
import contextlib
from fastapi import FastAPI
from weather_mcp import mcp

@contextlib.asynccontextmanager
async def lifespan(app: FastAPI):
 async with contextlib.AsyncExitStack() as stack:
 await stack.enter_async_context(mcp.session_manager.run())
 yield

app = FastAPI(lifespan=lifespan)

@app.get("/healthz")
def health() -> dict[str, bool]:
 return {"ok": True}

app.mount("/mcp", mcp.streamable_http_app())
```

## 6. `.dockerignore`

```
__pycache__/
*.pyc
.venv/
.git/
```

---

# Part B – Local Runs & Clients (45 min)

### 1. StdIO run

```
uv run python weather_mcp.py
```

### 2. HTTP run

```
uv run uvicorn app:app --host 0.0.0.0 --port 8000
MCP endpoint: http://localhost:8000/mcp
Health: http://localhost:8000/healthz
```

### 3. Inspector

```
npx @modelcontextprotocol/inspector
stdio target: python weather_mcp.py
http target: http://localhost:8000/mcp
```

### 4. Clients

- `client_stdio.py`

```

import asyncio
from mcp import ClientSession, StdioServerParameters, stdio_client

async def main() -> None:
 params = StdioServerParameters(command="python", args=["weather_mcp.py"])
 async with stdio_client(params) as (read, write):
 async with ClientSession(read, write) as session:
 await session.initialize()
 tools = await session.list_tools()
 print("TOOLS:", [tool.name for tool in tools.tools])

 alerts = await session.call_tool("get_alerts", {"state": "CA"})
 print("ALERT SAMPLE:", alerts.content[0].text[:400])

 forecast = await session.call_tool(
 "get_forecast", {"latitude": 38.58, "longitude": -121.49}
)
 print("FORECAST SAMPLE:", forecast.content[0].text[:400])

if __name__ == "__main__":
 asyncio.run(main())

```

◦ client\_http.py

```

import asyncio
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main() -> None:
 url = "http://localhost:8000/mcp"
 async with streamablehttp_client(url) as (read, write, _):
 async with ClientSession(read, write) as session:
 await session.initialize()
 tools = await session.list_tools()
 print("TOOLS:", [tool.name for tool in tools.tools])

 alerts = await session.call_tool("get_alerts", {"state": "TX"})
 print("ALERT SAMPLE:", alerts.content[0].text[:400])

 forecast = await session.call_tool(
 "get_forecast", {"latitude": 40.7128, "longitude": -74.0060}
)
 print("FORECAST SAMPLE:", forecast.content[0].text[:400])

if __name__ == "__main__":
 asyncio.run(main())

```

## 5. Contract smoke tests

- Create `tests/test_weather_mcp.py` that mounts `app:app` with `httpx.ASGITransport` and asserts deterministic tool payloads.
- Optional: `uv run schemathesis run http://localhost:8000/openapi.json --checks status_code_conformance --dry-run` to mirror Session 10 contract gates.

## Part C – Docker Hub + MCP Registry (45 min)

### Dockerfile

```
FROM python:3.12-slim AS builder
WORKDIR /app
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*
RUN curl -LsSf https://astral.sh/uv/install.sh | sh
ENV PATH="/root/.cargo/bin:$PATH"
COPY pyproject.toml ./
RUN uv venv && . .venv/bin/activate && uv pip install -e .

FROM python:3.12-slim
WORKDIR /app

ARG MCP_NAME=io.docker.zozo001.weather-mcp
LABEL io.modelcontextprotocol.server.name="${MCP_NAME}"

COPY weather_mcp.py ./weather_mcp.py
COPY app.py ./app.py
COPY --from=builder /app/.venv /app/.venv
ENV PATH="/app/.venv/bin:$PATH"
RUN useradd -m appuser
USER appuser

EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

### Build, run, and push (Docker Hub **zozo001**)

```
docker login --username zozo001
TAG=zozo001/weather-mcp:latest
MCP=io.docker.zozo001.weather-mcp

docker build -t "$TAG" --build-arg MCP_NAME="$MCP" .
docker run --rm -p 8000:8000 "$TAG"
docker push "$TAG"
```

### MCP Registry – **server.json**

```
{
 "$schema": "https://static.modelcontextprotocol.io/schemas/2025-07-09/server.schema.json",
 "name": "io.docker.zozo001.weather-mcp",
 "version": "0.1.0",
 "description": "Weather MCP demo (alerts + forecast tools)",
 "packages": [
 {
 "registry_type": "oci",
 "registry_base_url": "https://registry-1.docker.io",
 "identifier": "zozo001/weather-mcp",
 "version": "v0.1.0"
 }
]
}
```

Publish with the official CLI:

```
curl -L "https://github.com/modelcontextprotocol/registry/releases/download/v1.0.0/mcp-publisher_1.0.0_linux_amd64.tar.gz" | tar xz mcp-publisher
sudo mv mcp-publisher /usr/local/bin/mcp-publisher
mcp-publisher validate server.json
mcp-publisher login docker --username zozo001
mcp-publisher publish server.json
```

## Verify registry entry

```
curl "https://registry.modelcontextprotocol.io/v0/servers?search=io.docker.zozo001.weather-mcp"
```

---

## Remote Assistant Integration

- **Claude Desktop:** add a local MCP entry pointing to `python weather_mcp.py`.
- **Claude Custom Connector / LM Studio:** deploy the Docker image and expose `https://<host>/mcp`.
- **Inspector:** connect via stdio for debugging and HTTP for production parity.

---

## Troubleshooting

- Never print to stdout when running stdio (log to files instead).
- Ensure `USER_AGENT` is set or NWS rejects requests.
- If Docker push fails, re-run `docker login --username zozo001`.
- MCP Registry requires the Docker label `io.modelcontextprotocol.server.name` matching the `name` field.

---

## Student Success Criteria

- `python weather_mcp.py` responds to tool calls.
- `uvicorn app:app` serves MCP at `/mcp` and passes Inspector tests.
- `zozo001/weather-mcp:latest` runs locally and exposes `/mcp`.
- `server.json` validates and publishes to the MCP Registry.

---

## Remixing the Pattern for Movie Recommenders

Session 12 already has teams harden their REST endpoints (e.g., `/api/recommend-movie`). This workshop is the next logical mile: wrap that same logic in FastMCP so AI assistants can call it deterministically as `/tool/recommend-movie` while human clients still hit the HTTP API. The FastAPI app above shows the coexistence model—tool calls run through the MCP server, while `app.py` keeps your REST surface area alive. In practice:

1. Keep the recommendation core (vector search, collaborative filter, etc.) in a shared module.
2. Expose it twice—once via FastAPI/JSON just like EX3, and once via `@mcp.tool(name="recommend-movie")`.
3. Add it to the MCP streamable HTTP mount so connectors (Claude Desktop, Inspector, custom agents) discover it alongside `get_alerts`.

Encourage teams to remix this pattern so every movie recommender squad ships the same contract: an authenticated REST API for humans and `/tool/recommend-movie` for assistants, both powered by the exact same business logic.

## Example End-to-End Flow

1. A client POSTs to `/api/recommend-movie` with the viewer profile. FastAPI validates payloads and calls the shared recommender module.
2. Claude (or any MCP-aware assistant) invokes `/tool/recommend-movie` with the same schema. FastMCP calls the exact same shared module and streams the response back through MCP.
3. Both surfaces log to the same observability hooks, so product teams can compare assistant-initiated traffic vs human traffic without double maintenance.

## Shared Recommendation Core (`services/recommendation.py`)

```
from typing import Sequence

def recommend_movie(user_id: str, liked_titles: Sequence[str], mood: str) -> dict:
 # Vector search or rules live here; return structured data instead of strings
 return {
 "title": "Starship Rescue",
 "reason": f"Matches {mood} mood and overlaps with {liked_titles[:2]}",
 "alternatives": ["Nebula Knights", "Mission to Europa"],
 }
```

## FastAPI Endpoint (`api.py`)

```
from fastapi import APIRouter
from pydantic import BaseModel
from services.recommendation import recommend_movie

router = APIRouter(prefix="/api")

class RecommendRequest(BaseModel):
 user_id: str
 liked_titles: list[str]
 mood: str

@router.post("/recommend-movie")
def recommend(req: RecommendRequest):
 return recommend_movie(req.user_id, req.liked_titles, req.mood)
```

## MCP Tool (`movie_mcp.py`)

```
from mcp.server.fastmcp import FastMCP
from services.recommendation import recommend_movie

mcp = FastMCP("io.docker.zozo001.movie-mcp")

@mcp.tool(name="recommend-movie")
async def recommend_tool(user_id: str, liked_titles: list[str], mood: str) -> str:
 rec = recommend_movie(user_id, liked_titles, mood)
 return (
 f"Top pick: {rec['title']} - {rec['reason']}\n"
 f"Alternatives: {', '.join(rec['alternatives'])}"
)
```

## FastAPI + MCP Bridge ( app.py )

```
from fastapi import FastAPI
from api import router as api_router
from movie_mcp import mcp

app = FastAPI()
app.include_router(api_router)
app.mount("/mcp", mcp.streamable_http_app())
```

Now the exact same recommendation object feeds the REST route and the MCP tool:

- Humans call `POST /api/recommend-movie` .
- Assistants call `/tool/recommend-movie` (stdio or HTTP transport).
- Observatory parity stays aligned, and shipping MCP support becomes an additive change instead of a rewrite.

# AI Local SDK README – PydanticAI Gemini & Local Models

---

docs/sessions/optional/pydantic-ai-local-sdk-readme.md

This README replaces the PDF crash course with a Python-only walkthrough. It slots in after **Session 12 – Tool-Friendly APIs** (or during the EX3 sprint) to showcase a PydanticAI stack without touching OpenAI. We wire Google Gemini through the Generative Language API and provide a drop-in swap for local Ollama models.

- **Instructor GitHub handle:** `zozo123`
- **Model tracks:** Google GLA (Gemini 2.x) and Ollama-compatible local LLMs

Learners finish with:

1. A `uv`-managed Python project that wraps PydanticAI agents behind FastAPI + Typer.
2. Structured outputs enforced with `pydantic` models and retry logic handled by the agent.
3. Tool calls that blend first-party data (JSON/YAML) with lightweight HTTP lookups.
4. A toggleable provider layer ( `GOOGLE_API_KEY` vs `OLLAMA_BASE_URL` ) so the same code serves hosted or local models.

---

## Project Files (Python)

- `ai_local_sdk/settings.py`
- `ai_local_sdk/schemas.py`
- `ai_local_sdk/catalog.py`
- `ai_local_sdk/agent.py`
- `ai_local_sdk/service.py`
- `ai_local_sdk/api.py`
- `cli.py`

Every code block below is copy/paste ready and written in Python; shell snippets remain limited to environment setup.

---

## Before Workshop – Preflight (JiTT)

---

- Install `uv`  $\geq 0.5$  and Python 3.12+ (PydanticAI pins to `>=3.9`, we standardize on 3.12).
- Enable one model track:
  - **Google AI Studio:** create an API key at <https://aistudio.google.com/app/api-keys>, note the quota, and copy the key.
  - **Local:** install [Ollama](#), pull a model ( `ollama pull llama3.2` ) and confirm `ollama list`.
- `uv add` will grab extras; no `pip install --break-system-packages`.
- Clone or prep a repo with write access; the session assumes `git` is ready.

Environment secrets to export ahead of time:

```
export GOOGLE_API_KEY="..." # only if using Google GLA
export OLLAMA_BASE_URL="http://127.0.0.1:11434" # default daemon URL
```

## Agenda (3×45 Minutes)

Segment	Duration	Format	Focus
Bootstrap & providers	45 min	Talk + live coding	<code>uv</code> init, secrets, provider toggles
Typed agents & tools	45 min	Guided coding	Pydantic schemas, tool wiring, retries
Interfaces & ops	45 min	Pairing / solo	FastAPI surface, Typer CLI, local vs cloud runs

## Learning Objectives

- Explain how PydanticAI selects models ( `google-gla:` vs custom provider instances).
- Model structured responses with Pydantic types and force LLM retries until they validate.
- Register async tools that combine local datasets and live HTTP calls.
- Ship the agent behind FastAPI/CLI surfaces and switch between Google Gemini and Ollama without code rewrites.

## Part A – Foundation & Project Bootstrap (45 min)

### 1. Create the workspace

```
uv init ai-local-sdk --python 3.12
cd ai-local-sdk
uv add "pydantic-ai[google]" "google-generativeai" "pydantic-settings" "python-dotenv" \
 "httpx" "fastapi" "uvicorn[standard]" "typer[all]"
```

The `pydantic-ai[google]` extra supplies the Google Generative Language bindings. `google-generativeai` is optional but gives helper clients if you want raw fallback calls.

### 2. Scaffold project layout

```
ai-local-sdk/
├── pyproject.toml
├── .env.example
├── ai_local_sdk/
│ ├── __init__.py
│ ├── settings.py
│ ├── schemas.py
│ ├── catalog.py
│ ├── agent.py
│ ├── service.py
│ └── api.py
├── cli.py
└── README.md
```

### 3. Manage secrets

ai\_local\_sdk/settings.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
 model_config = SettingsConfigDict(env_file=".env", env_prefix="", extra="ignore")

 google_api_key: str | None = None
 gemini_model_id: str = "gemini-2.0-flash"

 use_local_model: bool = False
 ollama_base_url: str = "http://127.0.0.1:11434"
 local_model_id: str = "llama3.2"

 timeout_seconds: float = 45.0

settings = Settings()
```

.env.example

```
GOOGLE_API_KEY=replace-me
USE_LOCAL_MODEL=false
OLLAMA_BASE_URL=http://127.0.0.1:11434
LOCAL_MODEL_ID=llama3.2
GEMINI_MODEL_ID=gemini-2.0-flash
TIMEOUT_SECONDS=45.0
```

Keep `.env` out of source control; document the key exchange policy if learners need to rotate.

---

## Part B – Typed Agent & Tooling (45 min)

### 4. Model structured input/output

ai\_local\_sdk/schemas.py

```
from datetime import date
from pydantic import BaseModel, Field, PositiveInt, model_validator

class TripRequest(BaseModel):
 city: str = Field(min_length=2, description="City name to explore")
 days: PositiveInt = Field(le=7, description="Number of days to plan")
 budget_eur: int = Field(ge=100, le=5000)
 work_sessions: int = Field(ge=0, le=7, description="Focused work blocks needed")
 travel_date: date

 @model_validator(mode="after")
 def check_work_sessions(self) -> "TripRequest":
 if self.work_sessions > self.days:
 raise ValueError("work_sessions cannot exceed requested days")
 return self
```

```

class TripStop(BaseModel):
 title: str
 focus: str = Field(description="Key experience or takeaway")
 morning: str
 afternoon: str
 evening: str

class TripPlan(BaseModel):
 city: str
 summary: str
 remote_ready_score: int = Field(ge=1, le=10)
 highlights: list[str]
 days: list[TripStop]

```

The validator enforces `work_sessions <= days`, showing how Pydantic keeps LLM output usable.

## 5. Simple knowledge base

ai\_local\_sdk/catalog.py

```

from dataclasses import dataclass

@dataclass(slots=True)
class CityGuide:
 tagline: str
 coworking: list[str]
 meals: list[str]
 wifi_tip: str

CATALOG = {
 "lisbon": CityGuide(
 tagline="Atlantic light, historic hills, remote-friendly coffee",
 coworking=["Second Home Lisboa", "Resvés", "Avila Spaces"],
 meals=["Cervejaria Ramiro", "Time Out Market", "Prado"],
 wifi_tip="Ask for 'rede' passwords; most cafes post QR codes near the bar.",
),
 "barcelona": CityGuide(
 tagline="Beach mornings, Gothic afternoons, tapas evenings",
 coworking=["Cloudworks Passeig de Gràcia", "La Vaca", "OneCoWork Marina"],
 meals=["El Xampanyet", "Cal Pep", "Blai 9"],
 wifi_tip="Metro stations broadcast free Wi-Fi; grab an 8-day Hola BCN pass.",
),
}

```

Learners can extend this with their own data set or connect to Postgres later.

## 6. Build the agent

ai\_local\_sdk/agent.py

```

from dataclasses import dataclass

import httpx
from pydantic_ai import Agent, RunContext
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.models.openai import OpenAIChatModel
from pydantic_ai.providers.ollama import OllamaProvider

```

```

from .catalog import CATALOG
from .schemas import TripPlan, TripRequest
from .settings import settings

@dataclass(slots=True)
class PlannerDeps:
 http: httpx.AsyncClient

def _choose_model():
 if settings.use_local_model:
 provider = OllamaProvider(base_url=settings.ollama_base_url)
 return OpenAIChatModel(settings.local_model_id, provider=provider)
 google_provider = "google-gla" # Google Generative Language API (API key auth)
 return GoogleModel(settings.gemini_model_id, provider=google_provider)

planner_agent = Agent(
 model=_choose_model(),
 deps_type=PlannerDeps,
 output_type=TripPlan,
 instructions=(
 "You are a concise travel concierge for remote workers. "
 "Use registered tools, respect the requested number of days, "
 "and never hallucinate restaurants that are not in the catalog."
),
)

@planner_agent.tool
async def catalog_lookup(ctx: RunContext[PlannerDeps], city: str) -> dict:
 """Return coworking suggestions and food picks from the curated catalog."""
 guide = CATALOG.get(city.lower())
 if not guide:
 return {}
 return {
 "tagline": guide.tagline,
 "coworking": guide.coworking,
 "meals": guide.meals,
 "wifi_tip": guide.wifi_tip,
 }

@planner_agent.tool
async def nominatim_points(ctx: RunContext[PlannerDeps], city: str, query: str = "coworking") -> list[str]:
 """Search OpenStreetMap for venues (default: coworking spaces)."""
 url = "https://nominatim.openstreetmap.org/search"
 params = {"q": f"{query} {city}", "format": "json", "limit": 3}
 headers = {"User-Agent": "ai-local-sdk/0.1"}
 resp = await ctx.deps.http.get(url, params=params, headers=headers,
 timeout=settings.timeout_seconds)
 if resp.status_code != 200:
 return []
 return [
 f"{item['display_name']} · importance {float(item.get('importance', 0)):.2f}"
 for item in resp.json()
]

```

Swap Nominatim for any API learners can reach without extra keys; even a local JSON file works if outbound HTTP is blocked.

## 7. Service layer

ai\_local\_sdk/service.py

```
from contextlib import asynccontextmanager

import httpx

from .agent import PlannerDeps, planner_agent
from .schemas import TripPlan, TripRequest

@asynccontextmanager
async def deps_lifespan():
 async with httpx.AsyncClient() as client:
 yield PlannerDeps(http=client)

async def generate_plan(payload: TripRequest) -> TripPlan:
 async with deps_lifespan() as deps:
 result = await planner_agent.run(payload, deps=deps)
 return result.output
```

PydanticAI will retry automatically if the model returns invalid JSON, so learners can experiment with `TripPlan` fields and observe validation feedback in logs.

---

## Part C – Interfaces & Runs (45 min)

### 8. FastAPI surface

ai\_local\_sdk/api.py

```
from fastapi import FastAPI, HTTPException

from .schemas import TripPlan, TripRequest
from .service import generate_plan

app = FastAPI(title="AI Local SDK", version="0.1.0")

@app.post("/plan", response_model=TripPlan)
async def plan_trip(request: TripRequest) -> TripPlan:
 try:
 return await generate_plan(request)
 except Exception as exc:
 raise HTTPException(status_code=502, detail=str(exc)) from exc
```

Run it:

```
uv run uvicorn ai_local_sdk.api:app --reload --port 8031
```

Test the endpoint:

```
curl -s http://localhost:8031/plan \
-H "Content-Type: application/json" \
```

```
-d '{"city":"Lisbon","days":3,"budget_eur":1200,"work_sessions":2,"travel_date":"2025-06-10"}' \
| jq
```

## 9. Typer CLI for scripted use

cli.py

```
import asyncio
from datetime import date

import typer

from ai_local_sdk.schemas import TripPlan, TripRequest
from ai_local_sdk.service import generate_plan

cli = typer.Typer(help="Run the AI Local SDK agent from the terminal.")

@cli.command()
def plan(city: str, days: int = 3, budget: int = 1200, work: int = 1, when: date = date.today()):
 payload = TripRequest(city=city, days=days, budget_eur=budget, work_sessions=work,
 travel_date=when)
 plan_result: TripPlan = asyncio.run(generate_plan(payload))
 typer.echo(plan_result.model_dump_json(indent=2))

if __name__ == "__main__":
 cli()
```

Run:

```
uv run python cli.py plan --city "Barcelona" --days 4 --budget 1500 --work 2
```

## 10. Switching providers mid-session

- Google track:

```
cp .env.example .env
uv run uvicorn ai_local_sdk.api:app --reload
```

Open `.env`, keep `USE_LOCAL_MODEL=false`, and set `GOOGLE_API_KEY=***`. The agent will pick `google-gla: {GEMINI_MODEL_ID}` and authenticate with the key.

- Local track:

```
ollama pull llama3.2
export USE_LOCAL_MODEL=true
export LOCAL_MODEL_ID=llama3.2
export OLLAMA_BASE_URL=http://127.0.0.1:11434
uv run uvicorn ai_local_sdk.api:app --reload
```

`OllamaProvider` wraps an OpenAI-compatible client pointed at Ollama; no cloud calls involved.

## Extensions & Assessment

---

- Add a test that freezes the agent output by injecting a fake provider returning canned JSON; grade learners on schema validation.
- Use `planner_agent.with_history()` to persist chat state across multiple CLI calls.
- Introduce streaming via `planner_agent.stream(...)` and surface tokens over Server-Sent Events.
- Replace Eventbrite with a local SQLite `events` table to practice deterministic tool outputs.

**Check for understanding:** learners should demo both provider modes, explain how PydanticAI enforces structured output, and outline how to extend the agent with another tool (e.g., weather) without rewriting FastAPI.

# Streamlit + FastAPI Microservices Demo (Blockchain-Inspired)

---

docs/optional/blockchain-microservices-demo.md

## Learning Objectives

---

By completing this module, students will be able to:

1. **Explain** how distributed ledger systems maintain consistency without a central authority
2. **Demonstrate** the relationship between wallets, transactions, and blocks in a UTXO model
3. **Implement** REST APIs that coordinate state across multiple microservices
4. **Analyze** how proof-of-work mining creates consensus in competing chains
5. **Evaluate** trade-offs between centralized and decentralized system architectures

## Prerequisites

---

Students should have completed:

- FastAPI fundamentals (request/response models, async endpoints, background tasks)
- Streamlit basics (session state, form handling, live updates)
- Docker Compose essentials (multi-service orchestration, networking, volumes)
- Basic cryptography concepts (hashing, public/private key signatures)

**Estimated time:** 3-4 hours for core demo + 2 hours for optional fork/reorg extensions

---

## What we're building

---

- **Goal:** a three-service, all-HTTP lab that lets students watch "wallet → mempool → mined block" transitions without needing a real blockchain.
- **Stack:** Streamlit UI + two FastAPI services. Everything runs locally via `docker compose up`.
- **Story:** Students mint faucet coins, craft signed UTXO transactions, push them through a coordinator API, and trigger a proof-of-work toy miner to confirm blocks. The demo visualizes confirmations, forks, and "most accumulated work" decisions.

**Why this matters:** Real-world systems like Bitcoin, Ethereum, and enterprise blockchains all follow this pattern—wallets sign transactions, nodes maintain mempools, and miners/validators batch work into blocks. By implementing a toy version with familiar tools (FastAPI + Streamlit), students see how these pieces interact without drowning in cryptocurrency complexity.

---

## Scope guardrails for v1

---

- Keep the three services, but limit each to 1–2 source files so students can trace logic linearly.
- `node-miner` mines only when `/node/mine` is called. The handler builds a candidate block, brute-forces nonces (with a fixed attempt cap), and returns success/failure. No background miners, volumes, or reorg logic.

- `tx-coordinator` is a REST proxy with basic validation (non-empty inputs/outputs, simple signature placeholder). It forwards requests verbatim to the node and relays responses—no SSE, websockets, or event fan-out.
- `streamlit-wallet` is a single `app.py` with three tabs: Home, Wallets, Blocks & Mempool. It polls the coordinator every few seconds instead of subscribing to live streams.
- **Signature simplification:** Locking script = address string (no script interpreter). Signature = `sha256(txid + privkey)[:16]` (deterministic, verifiable without ECDSA overhead). This keeps v1 focused on transaction flow rather than cryptography edge cases.
- Rules are intentionally toy-like: PoW is "SHA-256 hash starts with N zero hex chars", and faucet amount is fixed.
- **Advanced features reserved for optional extension module:** Forks, Mining 101 playground with live sliders, drag-and-drop transaction builders, SSE/websockets, and multi-miner reorg demos move to a clearly labeled "Extension Module" roadmap so v1 stays teachable in one sitting.

## Teaching Progression (from simple → complex)

### Session 1: Single wallet, single transaction (30 min)

**Concepts introduced:** UTXO, transaction inputs/outputs, digital signatures

**Student actions:**

1. Create one wallet via Streamlit UI (generates keypair client-side)
2. Request faucet funds (POST to coordinator, which creates a coinbase UTXO)
3. View balance (GET from coordinator, which queries node's UTXO set)
4. Craft a transaction spending the faucet UTXO to a new address
5. Sign and submit (Streamlit signs locally, POSTs to coordinator)

**Teaching tips:**

- Use analogy: UTXO = a physical check made out to you; spending it = endorsing the check to someone else
- Show the JSON transaction structure on screen before signing so students see inputs referencing `prev_txid/prev_index`
- Emphasize that the coordinator validates the transaction *before* forwarding to the node (separation of concerns)

**Cognitive checkpoint** (before moving to Session 2):

- Ask students to predict: "If Alice has a 50-token UTXO and sends 30 to Bob, how many outputs will the transaction create?" (Answer: 2—one for Bob, one change back to Alice)
- Quick poll: "Where is your wallet balance stored?" Correct answer: "Nowhere—it's computed by summing UTXOs the node knows about"

**Common misconceptions:**

- "Balance is stored in the wallet" → No, balance = sum of UTXOs you can unlock with your private key
- "Transactions are instant" → No, they sit in the mempool until a miner includes them in a block

### Session 2: Mining & confirmations (30 min)

**Concepts introduced:** Mempool, blocks, proof-of-work, confirmations

**Student actions:**

1. Observe the pending transaction in the mempool table
2. Click "Mine Block" (sends POST `/api/mine` to coordinator, which forwards to the node)
3. Read the textual mining log the node returns (e.g., attempts count, winning nonce)
4. See the block appear in the Blocks table once a valid nonce is found
5. Wallet tab polls every few seconds and shows status change from "pending" to "confirmed (1)"

#### Teaching tips:

- Before starting the miner, ask: "Why can't we just add transactions to blocks immediately?"
- Explain the single difficulty knob: lower difficulty = faster demos, higher difficulty = more secure (real-world)
- Show that the block contains the transaction PLUS a coinbase reward for the miner
- Walk through the block header fields (prev\_hash links to parent, merkle\_root commits to all txs, nonce makes the hash valid)

**Visualization aid:** Streamlit prints log lines, e.g.

```
Target prefix: 0000
Tried 3,522 nonces → success with nonce 3521 (hash 0000c3a1...)
Block contains 1 user tx + coinbase reward
```

Reserve fancy sliders/graphs for the Extension Module.

#### Cognitive checkpoint (before moving to Session 3):

- Show two blocks with different difficulty prefixes (e.g., "000" took 50 attempts, "0000" took 8,000 attempts)
- Ask: "Why did adding one more zero roughly multiply attempts by 16?" (Answer: each hex digit has 16 possibilities —0-9, a-f)

### Session 3: Multiple wallets & change outputs (20 min)

**Concepts introduced:** Change addresses, transaction fees, UTXO consolidation

#### Student actions:

1. Create a second wallet ("Bob")
2. Send partial funds from first wallet to Bob (e.g., 30 of 50 tokens)
3. Observe that the transaction creates TWO outputs: 30 to Bob, 20 back as change
4. Mine a block to confirm
5. Both wallets now have new UTXOs they can spend

#### Teaching tips:

- Compare to paying with cash: "If you buy a \$3 coffee with a \$5 bill, you get \$2 in change"
- Show the UTXO tree: faucet → Alice(50) → [Bob(30), Alice\_change(20)]
- Introduce fees: "If outputs sum to less than inputs, miners keep the difference"

#### Cognitive checkpoint:

- Give students a scenario: "Alice has UTXOs of 10, 20, and 30 tokens. She wants to send 35 to Bob. Which UTXOs should she use?" (Answer: Multiple valid answers—discuss trade-offs between fee size and future spendability)

## Session 4 (Optional Extension Module): Forks & reorganizations (40 min)

**Concepts introduced:** Chain splits, accumulated work, consensus rules

**Prerequisite:** Students must complete Sessions 1-3 and pass a short quiz on UTXO mechanics before attempting fork demos.

### Student actions:

1. Enable the forked miner profile ( `docker compose --profile forked up` )
2. Submit a transaction and watch miners compete via CLI logs or a future Fork Visualizer tab
3. Observe two branches at the same height
4. See one branch overtake the other as more blocks accumulate
5. Watch the UI/logs highlight the reorg: orphaned blocks, transactions returning to mempool

### Teaching tips:

- Emphasize this is NOT an error state—it's how decentralized systems reach agreement
- Ask: "If Alice's payment appeared in the losing branch, is she still paid?" (Answer: No, tx goes back to mempool)
- Real-world parallel: Bitcoin/Ethereum forks happen regularly; exchanges wait 6+ confirmations to consider payments final

### Diagram for whiteboard:

```
Genesis → Block 1 → Block 2 → Block 3 (miner A) ← WINNING CHAIN (more work)
 ↘ Block 2' (miner B) ← ORPHANED
```

## Service Architecture (plain language first)

### Design philosophy

- **HTTP everywhere:** Pure REST + polling in v1 (no SSE/websockets). Students already know FastAPI, so the emphasis stays on clear request/response flows.
- **Clear responsibilities:** UI worries about user input, coordinator validates shapes, node enforces consensus rules. This separation makes debugging easier.
- **Observable state:** Every service exposes GET endpoints that return full state (mempool, UTXO set, chain tips) so students can inspect what's happening at each layer.

### Services (all speak HTTP)

1. `streamlit-wallet` (front-end)
  - Single-file Streamlit app acting as wallet + block explorer.
  - Generates keys client-side, stores them in session state, and polls the coordinator for balances, mempool, and blocks every few seconds.
  - Provides buttons for faucet funding, transaction submission, and "Mine Block" (which calls into the coordinator).

**Teaching angle:** This is the "user-facing app" like Coinbase or MetaMask. It knows your private keys but doesn't store the blockchain.

## 2. `tx-coordinator` (backend #1)

- FastAPI service that exposes a clean REST layer for the UI.
- Performs lightweight validation (non-empty inputs/outputs, sum checks, deterministic toy signatures) before forwarding requests to the node.
- No streaming features in v1; it simply proxies REST calls and surfaces node responses verbatim.

**Teaching angle:** This is the "business logic layer." Real companies add rate limiting, user authentication, and analytics here without touching the node code.

## 3. `node-miner` (backend #2)

- FastAPI service that owns the in-memory blockchain state, mempool, UTXO set, and simplified PoW miner loop.
- Provides HTTP endpoints for submitting transactions, inspecting chain tips, and adjusting a single difficulty value.
- Mines synchronously inside `POST /node/mine` : build candidate block from current mempool, brute-force nonce (single SHA-256 hash per attempt) until success or attempt cap; then append block, update UTXOs, prune mempool.

**Teaching angle:** This is the "consensus layer." In Bitcoin, thousands of these nodes run independently. If they disagree, the one with the most accumulated proof-of-work wins.

## 4. **Optional extra miners** (for fork demos)

- Exact copies of the node miner that we can spin up to simulate network competition.
- Marked as Phase 2 so v1 remains linear and approachable.

---

# Concept Glossary (introduce terms progressively)

---

## Core concepts (Phase 1)

- **UTXO (Unspent Transaction Output):** A single coin produced by a past transaction. If a UTXO says "5 tokens locked to Alice's address," only Alice (with her private key) can spend those 5 tokens, split them, or combine them in a new transaction.  
*Analogy:* Like a signed check made out to you—only you can deposit/endorse it.
- **Transaction input / output:** Inputs point to UTXOs you want to spend (like "I'm cashing this check"). Outputs describe new UTXOs you are creating (like "Write a new check to Bob for \$30"). Inputs must add up to at least as much value as the outputs (extra becomes a fee for the miner).
- **Digital signature:** Proves you own the private key for an address without revealing the key itself. Calculated using ECDSA (the same algorithm used in Bitcoin/Ethereum).

## Mining concepts (Phase 2)

- **Mempool:** Short for "memory pool." Every blockchain node keeps a list of valid-but-not-yet-mined transactions in RAM so miners can pick from them. Transactions wait here until a miner includes them in a block.  
*Note:* Yes, "mempool" is the standard Bitcoin/Ethereum term. We keep it for authenticity but always define it when teaching.

- **Block:** A batch of transactions plus a header that references the previous block. Together they form an append-only chain where every block depends on the one before it.  
*Components:* Header (prev\_hash, merkle\_root, timestamp, nonce) + List of transactions.
- **Proof of Work (PoW):** A lottery where miners keep hashing the block header with different nonces until the hash is smaller than a target number. Finding such a nonce is computationally hard (billions of attempts); verifying the result is one hash operation. More accumulated work = more security against rewrites.
- **Difficulty / target:** How small the winning hash must be. Smaller target → harder puzzle → longer expected mining time. In v1 we expose this as a single "required zero prefix" input so instructors can show that adding one zero roughly doubles expected work.
- **Confirmation:** Once your transaction sits in a block, each additional block on top counts as another confirmation. More confirmations make it exponentially harder for an attacker to rewrite history (they'd need to redo all the work in every block).  
*Rule of thumb:* Bitcoin exchanges wait 6 confirmations (~1 hour) before crediting deposits.

## Advanced concepts (Phase 4)

- **Fork / reorg:** When two miners find blocks at similar times, the chain temporarily splits. Eventually the branch with more accumulated work wins and the other branch is abandoned ("reorganized"). Transactions in orphaned blocks return to the mempool.  
*Why it matters:* Shows students that blockchain consensus is *probabilistic* not instant. This is why merchants wait for confirmations.
- **Accumulated work:** Sum of difficulty across all blocks in a chain. The chain with the most work is considered canonical. This prevents attackers from creating a longer chain by mining easy blocks quickly.

---

## Teaching Proof-of-Work (detailed lesson plan)

---

### Session 1: Console intuition (15 min)

**Materials:** Streamlit "Mine Block" log output

**Steps:**

1. Show the block header fields returned by the node ( `version` , `prev_hash` , `merkle_root` , `timestamp` , `difficulty_prefix` , `nonce` ).
2. Mine an empty block so the operation finishes quickly.
3. Read the textual log (attempt count, winning nonce, hash prefix) aloud.
4. Ask students what changed between attempts (only the nonce) to emphasize randomness.

**Key insight:** PoW = “keep hashing until SHA-256(header) starts with N zero hex chars.” No advanced math—just brute force.

### Session 2: Hands-on mining with one knob (20 min)

**Materials:** Blocks & Mempool tab (auto-refresh table)

**Steps:**

1. Submit a transaction so it enters the mempool.

2. Set difficulty prefix to `000` via Streamlit input → mine (fast success).
3. Set prefix to `0000` → mine again (notice longer runtime).
4. If max attempts are exceeded, discuss why the node aborts and how to retry.

#### Observation prompts:

- How many attempts did each run take? (Luck-driven.)
- Did transaction content change mining time? (No—difficulty dominates.)
- Why cap attempts per HTTP call? (Prevents a never-ending request.)

### Session 3: Confirmation depth (15 min)

**Materials:** Wallet + Blocks tabs

#### Steps:

1. After mining the Alice→Bob block, show the wallet balance and block entry.
2. Mine additional empty blocks to simulate time passing.
3. Point to the confirmation counter increasing with each new block.
4. Explain economic finality: the deeper the block, the harder it is to rewrite.

### Session 4: Fork & reorg (Phase 2)

Keep dual miners, fork visualizers, and live hash sliders for the optional module once the base experience is solid. When reintroduced, tie them to the same log-based intuition students built in v1.

#### Talking point checklist:

1. Mining = build block header, brute-force nonce until prefix rule passes.
2. Smaller prefix target ⇒ exponentially more work.
3. Verification is cheap (one SHA-256 hash).
4. Blocks chain via previous-hash pointers, so tampering breaks downstream hashes.
5. More confirmations mean more cumulative work to redo, which is why exchanges wait.

---

## Why This Fits the Course

### Pedagogical alignment

- **Builds on existing skills:** Students already know FastAPI (request handlers, Pydantic models) and Streamlit (forms, session state). This demo just adds blockchain *concepts* without new tools.
- **HTTP-first architecture:** Reinforces REST principles (stateless requests, resource-oriented endpoints, clear separation of concerns) without introducing message queues or gRPC complexity.
- **Observable systems:** Every service exposes GET endpoints for full state inspection—matches course emphasis on debuggable, testable code.
- **Progressive complexity:** Core demo (1 wallet, 1 tx, 1 block) works in 30 minutes. Fork/reorg extensions are optional advanced material.

## Real-world relevance

- Shows how microservices coordinate state via HTTP contracts (coordinator validates, node enforces consensus)
- Demonstrates deterministic PoW logic that can later be moved into background workers
- Introduces eventual consistency (transactions pending → confirmed, forks resolving over time)
- Connects to industry: students leave understanding why Coinbase waits for confirmations, how Ethereum gas fees work (tx fees in our demo), why "blockchain database" is usually the wrong tool

## Constraints met

- **Local-first:** Everything runs via `docker compose up`, no cloud accounts or external APIs
- **Minimal dependencies:** Just Python 3.12 + uv + Docker (students already have these from prior modules)
- **Time-bounded:** Core demo = 90 minutes lecture + 60 minutes lab. Fork extensions = optional 2-hour workshop.

## Wallet + Node Variations

---

### Flexibility for different class sizes

- **Small class (5-10 students):** Each student runs full stack locally, creates multiple wallets in their own Streamlit session
- **Medium class (15-30):** Instructor runs shared `node-miner` on classroom server, students connect Streamlit UIs to it (see tx from classmates in mempool)
- **Large class (30+):** Instructor pre-mines blocks with scripted transactions, students explore read-only via block explorer tabs

### Advanced extensions (post-lab exercises)

- **Multiple wallets per student:** Streamlit session state can hold keypairs for "Alice", "Bob", "Miner"—students act out multi-party scenarios
- **Faucet variations:** Instructor can fund specific students' addresses via `POST /node/tx` with a pre-signed coinbase
- **Scripted demos:** Python script calls coordinator API in a loop to simulate network activity (students see mempool fill/drain dynamically)
- **Competing miners:** Enable `docker compose --profile forked` to launch 2-3 node replicas—they race to mine blocks, demonstrating probabilistic consensus
- **Custom transaction types:** Advanced students can extend the UTXO model to support multi-sig, timelocks, or atomic swaps (concepts from real Bitcoin script)

---

## What Docker Compose Gives Us

---

### Single-command deployment

- `docker-compose.yml` spins up all three services plus a shared bridge network.
- Health checks ensure Streamlit waits for both FastAPI services to report ready before attempting to call them.
- Environment variables keep knobs adjustable (difficulty prefix, faucet amount, max attempts, miner address) without editing code.

## State management

- All chain data lives in memory inside `node-miner` — restarting the container resets the ledger (handy for rapid demos).
- Optional Phase 2 upgrade: add a persistent volume or JSON snapshot if instructors need continuity across sessions.

## Optional profiles

- `docker compose up` → core 3-service demo
- `docker compose --profile forked up` → adds `node-miner-2`, `node-miner-3` for fork visualizations
- `docker compose --profile debug up` → enables verbose logging and exposes pdb ports (future extension)

---

## HTTP Surface (high level)

Service	Endpoint	Method	Description	Teaching Note
streamlit-wallet → tx-coordinator	<code>/api/wallets</code>	POST	UI asks for new keypair+faucet; coordinator relays to node	"This is like clicking 'Create Account' in Coinbase"
streamlit-wallet → tx-coordinator	<code>/api/wallets/{address}</code>	GET	Fetch balance/UTXOs	"Shows spendable coins, not account balance"
streamlit-wallet → tx-coordinator	<code>/api/mempool</code>	GET	List pending transactions (poll every few seconds)	"Students can watch their tx enter/leave the queue"
streamlit-wallet → tx-coordinator	<code>/api/blocks</code>	GET	List most recent blocks	"Proof that mining captured the tx"
streamlit-wallet → tx-coordinator	<code>/api/tx</code>	POST	Submit signed transaction	"Streamlit signs locally (your keys never leave your browser)"
streamlit-wallet → tx-coordinator	<code>/api/mine</code>	POST	Trigger mining (for classroom control)	"In real Bitcoin, miners run 24/7 — here we control it for demos"
tx-coordinator → node-miner	<code>/node/tx</code>	POST	Validated tx enters mempool if UTXOs exist	"Coordinator already checked signature format; node checks UTXO set"
tx-coordinator → node-miner	<code>/node/utxos/{address}</code>	GET	Wallet balance/UTXO list	"Node is source of truth for what coins exist"
tx-coordinator → node-miner	<code>/node/chain/head</code>	GET	Returns tip hash, height, and latest block summary	"Used by Blocks tab to refresh every few seconds"

**Design rationale:** Coordinator = API gateway pattern. Keeps node focused on consensus logic while coordinator handles UI niceties (request validation now, SSE/fork visualizers later). Students learn to separate concerns across service boundaries.

---

## How the Pieces Work Together (step-by-step walkthroughs)

### 1. Faucet / onboarding

- Streamlit calls `POST /api/wallets`.
- Coordinator asks `node-miner` to mint a coinbase-style UTXO for that address and returns the txid.

- Streamlit triggers a manual refresh (or waits for the next poll) to show the new balance.

## 2. Spending

- Streamlit lists UTXOs via `GET /api/wallets/{addr}` and lets the student pick inputs + outputs.
- Streamlit signs locally (using `ecdsa`), shows the raw transaction JSON, and POSTs it.
- Coordinator recomputes the txid, rejects malformed scripts early, and forwards to the node; node validates against UTXO set and keeps it in the mempool.

## 3. Mining + confirmation tracking

- Student clicks “Mine Block,” coordinator forwards to node, and the node brute-forces nonces within the request.
- On success the node returns `{status:"mined"}` plus a summary (attempts, nonce, block hash); coordinator surfaces that text to the UI.
- Streamlit’s polling loop notices the mempool shrink and the new block entry; balances refresh automatically.

## 4. Fork + “most work” demo (Phase 2)

- When ready for advanced content, enable the `forked` compose profile or run two node-miner containers manually.
- Introduce a Fork Visualizer tab or CLI log to show competing branches and how the “most work” rule eventually collapses back to one chain.

# Build Specification (hand-off for automation)

## Repository layout

```
examples/blockchain-demo/
├── docker-compose.yml
├── .env.example
├── README.md (run instructions)
├── shared/
│ └── models.py # Pydantic/BaseModel schemas shared across services
├── node-miner/
│ ├── Dockerfile
│ ├── pyproject.toml # uv-managed (python 3.12)
│ └── app/
│ ├── main.py # FastAPI entrypoint
│ ├── mining.py # synchronous nonce search helpers
│ ├── chain.py # UTXO set, block store
│ └── settings.py
├── tx-coordinator/
│ ├── Dockerfile
│ ├── pyproject.toml
│ └── app/main.py
├── streamlit-wallet/
│ ├── Dockerfile
│ ├── pyproject.toml
│ └── app.py # Streamlit entrypoint
├── tests/
│ ├── test_transactions.py
│ ├── test_blocks.py
│ └── test_end_to_end.py # uses httpx.AsyncClient against both FastAPI apps
```

## Dependencies

- Python 3.12, managed via `uv`.

- Shared libs: `pydantic>=2.7` , `fastapi>=0.111` , `uvicorn[standard]` , `httpx` , `ecdsa` , `pysha3` (optional), `uvloop` (prod).
- Streamlit app: `streamlit>=1.36` , `httpx` (or `requests` ) for REST polling.

## Tooling & initialization

- Use `uv init --python 3.12 --package` inside each service folder ( `node-miner` , `tx-coordinator` , `streamlit-wallet` ) so every project pins Python 3.12 and generates a `pyproject.toml` , `.python-version` , and `.venv/` (ignored via `.gitignore` ).
- Shared models live outside of the `uv` projects; reference them by adding `path = "../shared"` entries in each `pyproject.toml` [ `tool.uv.sources` ] .
- Lock dependencies with `uv lock` and commit the resulting `uv.lock` .
- Standard commands:
  - `uv run fastapi dev app/main.py --port 8000` (node-miner)
  - `uv run fastapi dev app/main.py --port 8001` (tx-coordinator)
  - `uv run streamlit run app.py --server.port 8501` (Streamlit UI)
- When new packages are required use `uv add package-name` from the relevant service directory to keep lockfiles consistent.

## Dockerfiles

- Each service has its own Dockerfile at the service root ( `node-miner/Dockerfile` , etc.).
- **Important:** Shared models must be copied during Docker Compose build via build context, NOT via `COPY ../shared` inside Dockerfile.
- Pattern:

```
FROM python:3.12-slim AS runtime
RUN pip install --upgrade pip uv
WORKDIR /app
COPY pyproject.toml uv.lock ./
COPY app ./app
For Streamlit: COPY app.py ./
Shared models will be mounted at build time via compose context
COPY shared /app/shared
RUN uv sync --frozen
ENV PYTHONUNBUFFERED=1
CMD ["uv", "run", "fastapi", "run", "app/main.py", "--port", "8000"]
```

- Streamlit container swaps the final command for `["uv", "run", "streamlit", "run", "app.py", "--server.port", "8501", "--server.address", "0.0.0.0"]` .

## Docker Compose contract

- Services:
  - `streamlit-wallet` : builds from `./streamlit-wallet` , exposes `8501` .
  - `tx-coordinator` : builds from `./tx-coordinator` , exposes `8001` .

```
build:
 context: .
 dockerfile: ./streamlit-wallet/Dockerfile
```

```
build:
 context: .
```

```
dockerfile: ./tx-coordinator/Dockerfile
```

- `node-miner` : builds from `./node-miner` , exposes `8000` (all state in memory; restart resets chain).

```
build:
 context: .
 dockerfile: ./node-miner/Dockerfile
```

- Optional profile `forked` adds `node-miner-2` , `node-miner-3` containers reusing same image with env `NODE_NAME` .
- Networks: single bridge `blocknet` .
- Health checks: FastAPI endpoints `/healthz` , Streamlit requests `GET /_stcore/health` .
- Environment variables (load from `.env` ): `FAUCET_AMOUNT` , `MINER_ADDRESS` , `DEFAULT_DIFFICULTY_PREFIX` , `MAX_MINING_ATTEMPTS` , `STREAMLIT_API_BASE` .

## Shared models ( `shared/models.py` )

- **Simplified for v1** (no full ECDSA):
  - `HexBytes = Annotated[str, StringConstraints(pattern=r"^0x[0-9a-f]+$")]`
  - `Address = Annotated[str, StringConstraints(min_length=10, max_length=64)]`
  - `SimplifiedSignature = Annotated[str, StringConstraints(pattern=r"^0x[0-9a-f]{32}$")]` # 16-byte hex (deterministic)
  - `TxOut = BaseModel: value:int, locking_script:Address` # `locking_script` = recipient address
  - `TxIn = BaseModel: prev_txid:HexBytes, prev_index:int, unlocking_signature:SimplifiedSignature`
  - `Transaction = BaseModel: inputs:list[TxIn], outputs:list[TxOut], locktime:int=0`
  - `BlockHeader = BaseModel: version:int, prev_hash:HexBytes, merkle_root:HexBytes, timestamp:int, difficulty_prefix:str, nonce:int`
  - `Block = BaseModel: header:BlockHeader, transactions:list[Transaction], height:int`
  - `WalletInfo = BaseModel: address:Address, utxos:list[TxOut], balance:int`
  - `MiningResult = BaseModel: status:Literal["mined","no_solution"], attempts:int, nonce:Optional[int], block_hash:Optional[HexBytes], block:Optional[Block]`

**Signature validation rule** (implement in `tx-coordinator` and `node-miner`):

```
def validate_signature(tx: Transaction, input_idx: int, utxo: TxOut) -> bool:
 """
 V1 simplified signature: sha256(txid + privkey)[:32] as hex.
 For demo purposes, we can't verify without privkey, so coordinator
 just checks format. Node trusts coordinator's validation.

 Extension module will replace with proper ECDSA.
 """
 sig = tx.inputs[input_idx].unlocking_signature
 return bool(re.match(r"^0x[0-9a-f]{32}$", sig))
```

## API contracts

- **Node miner** ( `http://node-miner:8000` ):
  - `GET /healthz` → `{status:"ok"}`
  - `GET /node/chain/head` → `{height:int, tip:HexBytes, difficulty_prefix:str, latest_block:Block}`
  - `GET /node/blocks` → `{blocks:list[Block]}` (limit query param)

- `GET /node/mempool → {txs:list[Transaction], count:int}`
- `GET /node/utxos/{address} → WalletInfo`
- `POST /node/tx body Transaction → {txid:HexBytes, status:"accepted"|"rejected", reason?:str}`
- `POST /node/mine body {difficulty_prefix?:str, max_attempts?:int} → {status:"mined"|"no_solution", attempts:int, nonce?:int, hash?:HexBytes, block?:Block}`
- **Tx coordinator** (`http://tx-coordinator:8001`):
  - `POST /api/wallets → generates or requests wallet+faucet: {address:str, txid:HexBytes}`
  - `GET /api/wallets/{address} → WalletInfo`
  - `GET /api/mempool → proxied {txs:list[Transaction], count:int}`
  - `GET /api/blocks (optional limit) → proxied {blocks:list[Block]}`
  - `POST /api/tx body Transaction → {"txid":HexBytes,"status":"forwarded"}`
  - `POST /api/mine body {difficulty_prefix?:str, max_attempts?:int} → proxies to node and returns its log output.`
- **Streamlit client expectations:**
  - Reads API base URL from `STREAMLIT_API_BASE`.
  - Polls REST endpoints every 2–3 seconds for balances, mempool, and latest blocks.

## Streamlit UI features (v1)

- Tabs:
  - `Home` : quick instructions, buttons for "Create Wallet + Faucet" and "Mine Block", plus a textarea that displays the latest mining log returned by the coordinator.
  - `Wallets` : list stored keypairs (label, address, privkey), show UTXOs/balance for the selected wallet, simple form to build a transaction (select UTXO, enter recipient + amount, optional change address).
  - `Blocks & Mempool` : polling tables showing pending transactions and the last N blocks (hash, height, included txids). Optional auto-refresh toggle (default every 3 seconds).
- Use `st.session_state` to store wallets `{label,address,privkey}` and autopopulate the transaction form.
- Each tab starts with a short explainer (Markdown in an `st.expander`) so definitions stay close to the controls.
- **Error prevention UI patterns:**
  - Transaction form validates `sum(selected_utxos) >= sum(outputs)` client-side before allowing submission
  - Shows warning if no change output and inputs > outputs: "This will create a {fee} token miner fee. Continue?"
  - Disables "Mine Block" button if mempool is empty with tooltip: "Add transactions first"
  - Color-codes confirmation depth: red (0), yellow (1-2), green (3+)

**Extension module** adds more visualizations (Mining 101 slider, Fork Visualizer) once this core UI is solid.

---

## Troubleshooting Decision Tree (for instructors and TAs)

---

### Student reports: "My balance shows 0 after requesting faucet"

1. Check Streamlit UI refresh: Did they wait 3 seconds for polling or manually refresh?
2. Check coordinator logs: `docker compose logs tx-coordinator | grep /api/wallets`
  - If no POST logged → Streamlit form didn't submit correctly
  - If POST logged but no response → check node-miner logs
3. Check node-miner state: `curl http://localhost:8000/node/utxos/{address}`

- If returns empty → faucet transaction wasn't created
- If returns UTXO → UI polling issue, try manual refresh button

### Student reports: "Mining button does nothing"

1. Check mempool: `curl http://localhost:8000/node/mempool`
  - If count=0 → explain empty blocks are allowed, but demo needs a tx for visibility
2. Check max\_attempts: If difficulty is too high (e.g., "00000"), increase `MAX_MINING_ATTEMPTS` in `.env` or lower difficulty prefix
3. Check logs: `docker compose logs node-miner | grep "Mining"`
  - If no log lines → request didn't reach node, check coordinator proxy
  - If log shows "no\_solution" → explain retry with lower difficulty

### Student reports: "Transaction rejected: double-spend"

1. Explain: This is correct behavior! UTXO was already consumed.
2. Refresh wallet to see updated UTXO set after previous transaction was mined
3. Teaching moment: Show the UTXO lifecycle diagram (created → spent → removed from set)

### Student reports: "Confirmations not increasing"

1. Check if they're mining additional blocks: Confirmations only increment when NEW blocks build on top
2. Show block height: If stuck at height N, mine more blocks to see confirmations grow
3. Clarify: "Confirmation count = current\_height - block\_height\_containing\_tx"

---

## Assessment Rubric (for lab assignments)

---

### Knowledge (40 points)

- **Basic (28/40):** Can define UTXO, mempool, block, and proof-of-work in own words
- **Intermediate (34/40):** Explains why transactions need confirmations and how difficulty affects security
- **Advanced (40/40):** Compares UTXO model to account model, discusses trade-offs

### Skills (40 points)

- **Basic (28/40):** Successfully completes wallet creation, faucet request, and transaction submission
- **Intermediate (34/40):** Demonstrates change outputs, adjusts difficulty, interprets mining logs
- **Advanced (40/40):** Debugs failed transaction (e.g., insufficient funds), explains why signature validation failed

### Problem-Solving (20 points)

- **Basic (14/20):** Follows step-by-step guide without errors
- **Intermediate (17/20):** Recovers from one error independently (e.g., refreshes UI when balance doesn't update)
- **Advanced (20/20):** Identifies root cause of issue using logs/curl commands without instructor help

### Partial credit guidelines:

- Attempted transaction that fails validation → 50% credit if student explains what was wrong
- Incorrect explanation of UTXO but correct demonstration → 70% credit (doing > saying)
- Mining fails due to difficulty too high but student troubleshoots → full credit for problem-solving section

### Extension Module Bonus (20 points)

- Successfully triggers fork scenario → +10 points
- Correctly predicts which chain will win before simulation completes → +5 points
- Explains accumulated work calculation with example → +5 points

**Total possible:** 100 points (+ 20 bonus for extension module)

---

## Common Student Questions (FAQ for instructors)

### Q: "Why use UTXOs instead of account balances like my bank?"

A: Bitcoin's UTXO model makes double-spending impossible—once you spend a UTXO, it's gone forever. Ethereum uses accounts for simplicity, but requires more complex state management. Both models are valid; we teach UTXO first because the constraints make correctness easier to reason about.

### Q: "Can I mine blocks faster by running more Docker containers?"

A: Not in this v1 demo—only one miner runs per node container and mining is synchronous. But in real blockchains, yes! More machines = more hashpower = higher chance of winning the race. The extension module demonstrates this with competing miners.

### Q: "What if I sign a transaction but never broadcast it?"

A: No one knows you created it. Transactions only exist once miners see them. (Analogy: writing a check but never handing it to the recipient.) This is why blockchain wallets can "draft" transactions offline.

### Q: "Why do we need mining at all? Can't the node just append transactions immediately?"

A: Who decides the order? In a distributed system with no central authority, mining is the *lottery* that prevents anyone from unilaterally controlling the chain. The cost (electricity/computation) deters spam and Sybil attacks. This is the core innovation of Bitcoin.

### Q: "Is this code production-ready?"

A: No—it's a teaching tool. Real blockchains add: peer-to-peer networking, disk persistence, consensus edge cases (orphan handling, reorg limits), proper ECDSA signature verification, script interpreters, and cryptographic accumulators for UTXO sets.

### Q: "Can I use this for my cryptocurrency project?"

A: Please don't. Use established libraries (bitcoinlib, web3.py) or frameworks (Substrate, Cosmos SDK). This demo intentionally simplifies security-critical components (deterministic signatures, no replay protection, no nonce/sequence numbers). See the "Why This Matters" section for pointers to production systems.

### Q: "My transaction disappeared from the mempool without being mined!"

A: This shouldn't happen in v1 (no reorgs). Check if node restarted (loses in-memory state). Extension module handles this via reorgs. For now, resubmit the transaction.

---

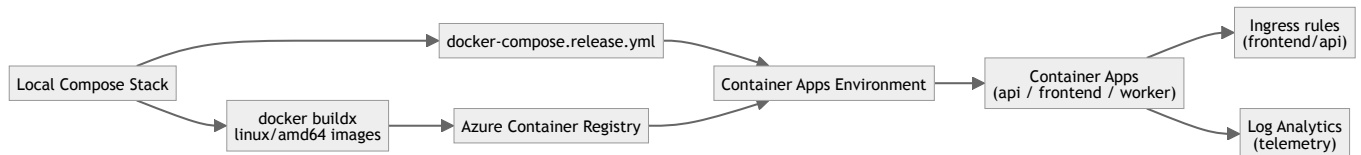
### Document changelog:

- v2.1: Added cognitive checkpoints, troubleshooting decision tree, partial-credit rubric, simplified signature model, fixed Docker build context
- v2.0: Restructured with explicit learning objectives, progressive teaching phases, FAQ section
- v1.0: Original implementation-focused spec

# Azure Container Apps Deployment Playbook

docs/deployments/azure-container-apps.md

This guide walks EX3 teams through deploying their Docker Compose stack (FastAPI API, Redis, nginx/frontends, and optional extras) to Azure Container Apps using the Azure for Students credit. Follow it end-to-end so graders can browse your project live without you incurring charges.



## 1. Prerequisites

- Azure for Students account with active \$100 credit.
- CLI tools installed locally:
  - `az` (Azure CLI) logged in with `az login`.
  - `docker`, `docker compose`, and `docker buildx`.
- A Compose stack that runs locally: `docker compose up --build` succeeds.
- Each service has a Dockerfile and can be built for `linux/amd64`.
- No secrets committed to Git; use environment variables or Azure secrets instead.

## 2. Prepare a Release Compose File

1. Keep your development file (e.g. `docker-compose.yml`) as-is.
2. Create `docker-compose.deploy.yml` that replaces every `build:` block with an image reference (`<acr>.azurecr.io/service:tag`) and strips dev-only mounts.
3. Generate the final release file:

```
docker compose -f docker-compose.yml -f docker-compose.deploy.yml config \
> docker-compose.release.yml
```

Only include published ports for services that must be reachable from a browser (typically the nginx or frontend gateway).

## 3. Define Azure Variables

Pick values once and reuse them. Keep them in a shell script or `.env` that you `source` locally (never commit real secrets).

```
export SUBSCRIPTION="<your-subscription-id>"
export LOCATION="westeurope"
export RESOURCE_GROUP="eass8-ex3-rg"
```

```
export ACR_NAME="eass8acr$RANDOM" # must be unique, lowercase
export ENV_NAME="eass8-containerapps"
```

Update the subscription context:

```
az account set --subscription "$SUBSCRIPTION"
```

---

## 4. Provision Azure Infrastructure

---

```
az provider register --namespace Microsoft.App
az provider register --namespace Microsoft.OperationalInsights
az provider register --namespace Microsoft.ContainerRegistry

az group create -n "$RESOURCE_GROUP" -l "$LOCATION"

az acr create -n "$ACR_NAME" -g "$RESOURCE_GROUP" \
 --sku Basic --admin-enabled true
ACR_LOGIN_SERVER=$(az acr show -n "$ACR_NAME" -g "$RESOURCE_GROUP" \
 --query loginServer -o tsv)

az monitor log-analytics workspace create \
 -g "$RESOURCE_GROUP" -n "${RESOURCE_GROUP}-log" -l "$LOCATION"
LAW_ID=$(az monitor log-analytics workspace show \
 -g "$RESOURCE_GROUP" -n "${RESOURCE_GROUP}-log" \
 --query customerId -o tsv)
LAW_KEY=$(az monitor log-analytics workspace get-shared-keys \
 -g "$RESOURCE_GROUP" -n "${RESOURCE_GROUP}-log" \
 --query primarySharedKey -o tsv)

az containerapp env create \
 -g "$RESOURCE_GROUP" -n "$ENV_NAME" -l "$LOCATION" \
 --logs-workspace-id "$LAW_ID" \
 --logs-workspace-key "$LAW_KEY"
```

---

## 5. Build and Push Images

---

Azure Container Apps currently runs on `linux/amd64`. If you are on Apple Silicon, force that platform.

```
docker buildx create --name eass8 --use || docker buildx use eass8

Example for three services; adjust paths, tags, and Dockerfiles
docker buildx build --platform linux/amd64 \
 -t "$ACR_LOGIN_SERVER/api:1" \
 -f services/api/Dockerfile services/api --push

docker buildx build --platform linux/amd64 \
 -t "$ACR_LOGIN_SERVER/recommender:1" \
 -f services/recommender/Dockerfile services/recommender --push

docker buildx build --platform linux/amd64 \
 -t "$ACR_LOGIN_SERVER/frontend:1" \
 -f services/frontend/Dockerfile services/frontend --push
```

Update `docker-compose.deploy.yml` with the pushed image tags.

---

## 6. Install Compose Extension and Deploy

---

```
az extension add --name containerapp --upgrade
az extension add --name containerapp-compose --upgrade

ACR_USER=$(az acr credential show -n "$ACR_NAME" \
 --query username -o tsv)
ACR_PASS=$(az acr credential show -n "$ACR_NAME" \
 --query passwords[0].value -o tsv)

az containerapp compose create \
 --resource-group "$RESOURCE_GROUP" \
 --environment "$ENV_NAME" \
 --location "$LOCATION" \
 --compose-file docker-compose.release.yml \
 --registry "$ACR_LOGIN_SERVER" \
 --registry-username "$ACR_USER" \
 --registry-password "$ACR_PASS"
```

Each Compose service becomes an individual Container App instance inside the shared environment.

---

## 7. Open Ingress and Set Secrets

---

Expose only the endpoint you want graders to hit (usually `frontend` or `nginx`). Internal services communicate over the automatically created internal network.

```
Example: expose frontend on port 80
az containerapp update \
 -g "$RESOURCE_GROUP" -n frontend \
 --ingress external --target-port 80 \
 --min-replicas 1 --max-replicas 2

Optional: expose API directly for debugging
az containerapp update \
 -g "$RESOURCE_GROUP" -n api \
 --ingress external --target-port 8000 \
 --min-replicas 1 --max-replicas 2
```

Set secrets and environment variables without hardcoding them in the Compose file:

```
az containerapp secret set \
 -g "$RESOURCE_GROUP" -n api \
 --secrets db_url="postgresql://user:pass@host:5432/db"

az containerapp update \
 -g "$RESOURCE_GROUP" -n api \
 --set-env-vars DATABASE_URL=secretref:db_url
```

Repeat for any other sensitive values (JWT keys, API keys, etc.).

---

## 8. Wire the Frontend to the API

---

Fetch the public FQDNs:

```
FRONTEND_URL=$(az containerapp show \
 -g "$RESOURCE_GROUP" -n frontend \
 --query properties.configuration.ingress.fqdn -o tsv)
API_URL=$(az containerapp show \
 -g "$RESOURCE_GROUP" -n api \
 --query properties.configuration.ingress.fqdn -o tsv)
```

If the frontend needs the API URL, set it as an environment variable:

```
az containerapp update \
 -g "$RESOURCE_GROUP" -n frontend \
 --set-env-vars VITE_API_BASE="https://$API_URL"
```

Enable CORS in FastAPI for `https://$FRONTEND_URL`. If you do not want to touch code, route API calls through an nginx reverse proxy in the frontend container.

---

## 9. Validate the Deployment

---

1. Visit `https://$FRONTEND_URL` in a browser—ensure the UI loads and can call the API.
2. Run automated smoke tests from your CI or local machine pointing to the cloud URLs.
3. Inspect logs if something fails:

```
az containerapp logs show \
 -g "$RESOURCE_GROUP" -n api --follow
```

4. Screenshot the application running and capture the `az containerapp show` outputs for submission evidence.
- 

## 10. Document for Graders

---

Your EX3 README must include:

- The full set of commands above with your actual resource names.
  - The public URL(s) and any demo credentials.
  - A link to an unlisted YouTube walkthrough (≤3 min) showing the live Azure deployment responding to real requests.
  - A rollback/cleanup checklist (below) so graders know you are staying within the free tier.
  - Notes about monitoring the credit balance ( `Cost Management + Billing` → `Cost analysis` ).
- 

## 11. Tear Down (After Demo Week)

---

```
az group delete -n "$RESOURCE_GROUP" --yes --no-wait
```

This removes the Container Apps environment, the ACR, and the Log Analytics workspace—preventing accidental credit usage.

---

## 12. Troubleshooting Checklist

---

- **Ingress returns 404:** verify the target port matches the container listener ( `uvicorn` defaults to 8000).
- **Image fails to start:** use `az containerapp revisions list` and `az containerapp revision show` to read container logs and exit codes.
- **Apple Silicon crash:** rebuild images with `--platform linux/amd64` and push again.
- **Secret not found:** double-check the `secretref:` syntax and that the secret name matches exactly.
- **Compose feature unsupported:** Container Apps ignores `depends_on` , `profiles` , and volume mounts. Replace mounts with Azure File shares or bake data into the image when necessary.

Document any extra fixes you needed so your teammates (and graders) can reproduce the deployment quickly.

# AI-Assisted Coding Workflow (Codex Paradigm)

docs/workflows/ai-assisted/README.md

This repository now follows the workflow explained in Anindya Chakraborty’s essay, *AI Assisted Coding: Quicker Code Doesn’t Mean Higher Velocity* (Nov 9, 2025). The paper’s core claim is that AI agents shifted the bottleneck **from writing code to understanding, testing, and reviewing it**. To keep course materials and exercises honest we:

- 1. Plan each change in teacher-friendly, reviewable slices.
- 2. Generate code with assistants only inside those slices.
- 3. Split and annotate the output so humans can understand it quickly.
- 4. Run deterministic verification before asking for peer review.
- 5. Teach the above loop explicitly so students can reuse it outside the course.

This directory documents the paradigm Codex follows when touching this repo.

## Lifecycle Overview

Loop	Human-owned questions	Assistant-owned tasks	Teaching hook
Scope & Spec	What problem are we solving? Where does it sit in the roadmap?	Draft initial outline/prompts once constraints are known.	Use Session 03 storyboard format; stress “one spec file per logical slice.”
Prompt & Generate	Define shape of the change (files, APIs, invariants).	Produce candidate code/tests/docs for the slice.	Model “drive, don’t delegate” pairing during live coding.
Split & Annotate	Group diffs into ≤150-line hunks, explain intent.	Suggest chunking + summary text.	Use the repo’s splitter/explainer workflow before PRs.
Review & Transfer	Validate maintainability, reject magic numbers, note risks.	Provide auto-generated review notes per hunk.	Run review checklist in Part C of labs.
Verify & Publish	Run pytest/ruff/mypy, re-read docs, ship.	Trigger local commands, highlight failures.	Every lab ends with the verification block already in the session doc.

## Directory Structure

```
workflows/
└─ ai-assisted/
 │ README.md # Overview (this file)
 │ teaching-guide.md # How instructors narrate & grade the paradigm
 │ repo-structure.md # How this repo enforces the paradigm (spec folders, logs)
 │ templates/
 │ │ feature-brief.md # Template for “small, reviewable slice” briefs
 │ │ checklists/
 │ │ │ review.md # Review + verification checklist tied to the PDF advice
```

## Using These Files

- 1. **Before prompting** – fill in `templates/feature-brief.md` so the scope stays under 150 LOC and the assistant knows the constraints.
- 2. **After generation** – run the splitter (or summarize manually) and attach annotations taught in `teaching-guide.md`.
- 3. **During review** – read `checklists/review.md` aloud with students; the PDF is explicit that reviewers must spend time understanding the change, so we make the questions concrete.

4. **During instruction planning** – `repo-structure.md` links each session doc to the workflow so the pedagogy stays consistent.

**Reminder:** AI-written code is treated like external contributions. Assume nothing about correctness or maintainability until the checklists pass.

# Repo Structure Changes for the AI-Assisted Paradigm

docs/workflows/ai-assisted/repo-structure.md

The repo now encodes the workflow outlined in the “AI Assisted Coding: Quicker Code Doesn’t Mean Higher Velocity” PDF. This document points to the key spots.

## Reference Sources

- Medium article: [AI Assisted Coding: Quicker Code Doesn’t Mean Higher Velocity](#).
- docs/workflows/ai-assisted/ — this folder with guides, templates, and checklists (see README.md).

## Mandatory Artifacts per Change

1. **Feature Brief** (stored anywhere under docs/ or directly in the PR) using templates/feature-brief.md — proves the change was scoped before prompting.
2. **Split/Annotation Notes** — either produced via armchr or summarized manually, but always referenced in commits/PRs.
3. **Checklist Run** — reviewers confirm each item in checklists/review.md before merging.
4. **Verification Log** — reference the commands from the relevant session doc (usually uv run pytest + a curl/script). Add the output summary to the PR description.

## Directory Pointers

```
docs/
├── sessions/ # canonical teaching scripts
├── reference/ai-assisted-*.txt # verbatim article excerpt for quick quoting
├── workflows/
│ └── ai-assisted/
│ ├── README.md
│ ├── teaching-guide.md
│ ├── repo-structure.md
│ ├── templates/
│ │ └── feature-brief.md
│ ├── checklists/
│ │ └── review.md
```

## Session Doc Hooks

- Every session that introduces non-trivial code now links to the workflow section in its “Before Class” or “Part B” steps. When updating a session file, add a line like:  
“Scope this change using workflows/ai-assisted/templates/feature-brief.md before prompting your assistant.”
- Session 08’s AI lab should cross-link to this folder explicitly; see the TODO note inside that file.

## Git Hygiene Expectations

- Keep commits ≤150 LOC in touched files unless impossible (document why in the brief when that happens).
- Reference the brief ID in commit messages: feat: repo-db brief-004 apply sqlite repository.

- Attach the checklist results ( `[x] Tests` , `[x] Review` , `[x] Split/Explain` ) to PR templates.

## Tooling Integration

---

- Adopt `armchr` locally (documented in `docs/workflows/ai-assisted/README.md` ) to split large diffs. Until an automated integration lands, copy the annotations into PR descriptions.
- Encourage `git worktree` usage for parallel AI sessions, mirroring the Reddit workflow quoted in the Medium article comments.

# Teaching Guide — AI-Assisted Workflow

docs/workflows/ai-assisted/teaching-guide.md

This guide translates the PDF’s recommendations into classroom moves. Every time Codex (or an instructor) touches the repo, we model the same habits so students internalize them.

## Learning Goals

1. **Expose the new bottleneck** — students must see that code generation is cheap but understanding/testing is not.
2. **Practice working in reviewable slices** — no feature merges without a filled `feature-brief.md`.
3. **Normalize shared ownership** — both “author” and “reviewer” learn the code together, so we rely on annotations + checklist items to keep cognitive load manageable.
4. **Use AI as a pair, not an autopilot** — narrate the architecture first, then let the assistant fill in code under those constraints.

## 3-Block Lesson Structure

Each 45-minute block already has A/B/C structure. Layer the workflow like this:

Session Block	What to model	Tools/Docs
Part A (theory)	Draw the “old vs new bottleneck” chart from the PDF. Highlight why the repo has specs + checklists.	<code>workflows/ai-assisted/repo-structure.md</code> diagrams
Part B (build)	Live-fill <code>templates/feature-brief.md</code> , show the exact prompt you send to Codex/Claude, watch the diff, then split + annotate.	Template + <code>armchr</code> splitter (or manual grouping)
Part C (deep dive)	Run through <code>checklists/review.md</code> with students reviewing the Part B change. End with verification commands from the session doc.	Review checklist + existing lab verification

## Session Mapping

- **Sessions 03–05:** Start using feature briefs when migrating FastAPI → SQLAlchemy → Postgres. Emphasize “one slice per subsystem” (config, models, repo, tests).
- **Session 06:** Show how the same workflow applies to UI additions; spec the dashboard API surface first.
- **Session 07:** Tie the checklist to logging/tests so students see review + observability alignment.
- **Session 08:** Make this the capstone on AI pair programming—compare a “big prompt” vs “brief-driven slices” and measure review time.
- **Sessions 09–12:** Treat refactors/ops work the same way (brief + checklist + verification) so the habit persists.

## Classroom Prompts

1. **“Scope call”** — ask students to read the brief aloud before code generation. Partners must restate success criteria.
2. **“Split & tell”** — after the assistant produces code, students mark each logical chunk ( $\leq 150$  LOC) and summarize what changed before anyone reviews.
3. **“Checklist stand-up”** — use the review checklist as a mini-retrospective; every student answers one bullet.

4. **“Ownership swap”** — have the reviewer explain the change back to the author; if they struggle, the annotations are insufficient.

## Teaching Tips

---

- Remind everyone that AI-written tests are not automatically trustworthy. Pair students so one writes/validates the tests while the other drives the implementation prompt.
- Celebrate when students delete or rewrite AI output. The PDF’s thesis is that understanding + maintainability trump raw LOC.
- Keep diffs small even in demos. If a live coding segment risks going beyond 150 LOC, pause and defer part of the change to the next block.
- Reference the article directly—drop the Medium link in session slides/notes so students can read the rationale outside class.

# Review Checklist (AI-Assisted Workflow)

---

docs/workflows/ai-assisted/checklists/review.md

Inspired by *AI Assisted Coding: Quicker Code Doesn't Mean Higher Velocity*.

## Before Requesting Review

---

- ☐ Feature brief is attached and scoped to a single logical change.
- ☐ Diffs are split/annotated ( $\leq 150$  LOC chunks) — include armchr notes or manual summaries.
- ☐ Tests/linters listed in the brief all ran locally with results noted.
- ☐ Documentation/session scripts updated if behavior changed.
- ☐ Verified that assistant-written tests actually fail when the code is broken (spot-check one test by mutating the code).

## During Review (Author + Reviewer together)

---

- ☐ Walk through each chunk, restating intent to confirm shared understanding.
- ☐ Trace critical paths manually (no relying solely on autogenerated tests).
- ☐ Confirm there's no hidden hard-coding or silent error handling added by the assistant.
- ☐ Ensure both author and reviewer can explain the change without looking at the diff — if not, add annotations/cleanup.
- ☐ Treat the code like an external contribution: assume bugs until proven otherwise.

## Before Merge

---

- ☐ Reviewer signs off that maintainability is acceptable (naming, duplication, logging).
- ☐ Checklist items copied into the PR template with `[x]` marks.
- ☐ Verification commands pasted (or referenced) with timestamps.
- ☐ Follow-up briefs filed for anything deferred.

# Feature Brief Template

---

docs/workflows/ai-assisted/templates/feature-brief.md

Copy this into your PR or `docs/notes/` folder before prompting the assistant. Keep each brief scoped to a reviewable slice ( $\leq 150$  LOC touched per file set).

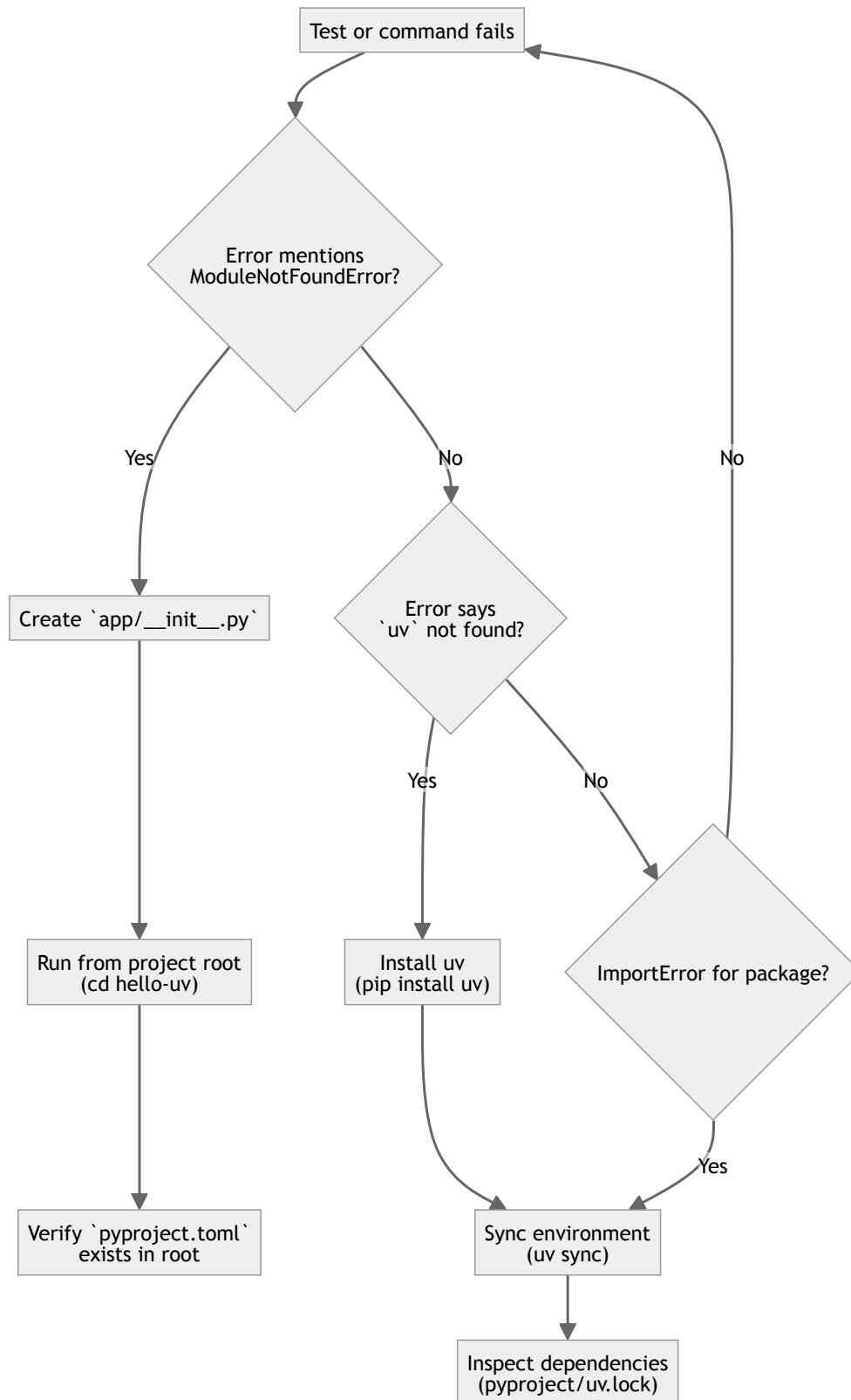
- **Brief ID:** `brief-###`
- **Date / Session:** e.g., `2026-03-23 · Session 04`
- **Owner(s):** name(s) or pair
- **Intent (2 sentences max):** What outcome will this slice deliver?
- **Trigger:** link to exercise requirement, bug ticket, or session step.
- **Files/Areas allowed:** list concrete modules/files.
- **Non-goals:** what must *not* change in this slice?
- **Assistant prompt outline:** bullet list of instructions you will feed into Codex/Claude (order matters).
- **Verification commands:** enumerate the exact `uv run ...` steps and curl checks.
- **Review focus:** what should reviewers scrutinize? (edge cases, migrations, etc.)
- **Docs/Teaching update:** any session/README text that must be edited alongside the code.

When done, attach the completed brief to the PR and reference it in commit messages.

# EASS Course Troubleshooting Guide

docs/troubleshooting.md

## General Environment & Setup



## "ModuleNotFoundError: No module named 'app'"

### Symptoms:

```
Traceback (most recent call last):
 File "tests/test_movies.py", line 3, in <module>
 from app.main import app
ModuleNotFoundError: No module named 'app'
```

### Causes:

1. Missing `app/__init__.py` (package marker)
2. Running tests from wrong directory
3. Project root missing `pyproject.toml`

### Fixes:

```
Ensure app/__init__.py exists (can be empty)
touch app/__init__.py

Run tests from project root (where pyproject.toml is)
cd /path/to/hello-uv
uv run pytest tests/

Verify project structure
ls -la | grep -E "(pyproject.toml|app|tests)"
```

---

## "uv command not found"

### Fixes:

```
Install uv globally
pip install uv

Or use Python module syntax
python -m uv --version

Check installation
which uv
```

---

## Dependencies not installing

### Symptoms:

- `ImportError: No module named 'fastapi'`
- `ModuleNotFoundError: No module named 'sqlmodel'`

### Fixes:

```
Sync dependencies
uv sync
```

```
Add missing package
uv add fastapi

Check installed packages
uv pip list

Verify pyproject.toml has dependencies
cat pyproject.toml | grep -A 20 "dependencies"
```

---

## Docker Issues

---

### "Docker command not found"

#### Fixes:

- Install Docker Desktop for macOS or Windows
- On WSL, ensure Docker Desktop is running: `docker ps`

---

### "Cannot connect to Docker daemon"

#### On macOS/Windows:

```
Open Docker Desktop application, wait for whale icon to settle
open -a Docker

Then retry
docker ps
```

#### On WSL:

```
Ensure Docker Desktop is running on Windows host
Then in WSL terminal:
docker ps
```

---

### "docker compose up fails" or "command not found: docker-compose"

#### Modern fix (Docker >= 2.3):

```
Use new syntax (not dash)
docker compose up

NOT: docker-compose up (old)
```

#### If you have old Docker:

```
Install compose plugin
docker plugin install compose
```

```
Or upgrade Docker Desktop
```

## vLLM on macOS is very slow or fails to pull

**Context:** vLLM's official image (`vllm/vllm-openai:latest`) targets Linux/amd64 and is GPU-optimized. On Apple Silicon it runs under emulation without GPU and can be slow.

### Fixes / Tips:

```
Force amd64 platform and use a tiny model to reduce memory/CPU load
docker run --rm -it \
 --platform=linux/amd64 \
 -p 8000:8000 \
 vllm/vllm-openai:latest \
 --host 0.0.0.0 --port 8000 \
 --model TinyLlama/TinyLlama-1.1B-Chat-v1.0
```

- Prefer LM Studio for Apple Silicon if you only need a local test endpoint.
- If the image pull is too large for the network, skip vLLM for the lab and use LM Studio instead (Session 08).
- For gated models, set `-e HF_TOKEN=...` and mount `~/.cache/huggingface` into the container to avoid repeat downloads.

## “LM Studio is blocked” or “Need a CLI-only LLM endpoint”

**Symptom:** Lab machines cannot install LM Studio or Docker images.

**Quick fix:** Use `llama.cpp` with the Gemma 3 270M Instruct GGUF hosted from Hugging Face. Works entirely in terminal windows and mimics the OpenAI API.

```
1) Install binaries
brew install llama.cpp

2) Launch server (pulls weights once)
llama-server \
 -hf ggml-org/gemma-3-270m-it-GGUF \
 --model gemma-3-270m-it \
 --port 8080 \
 --host 127.0.0.1 \
 --jinja \
 -c 4096

3) Test OpenAI-compatible endpoint
curl http://127.0.0.1:8080/v1/chat/completions \
 -H "Content-Type: application/json" \
 -d '{
 "model": "gemma-3-270m-it",
 "messages": [
 {"role": "user", "content": "Say hello and tell me your parameter count."}
]
 }'
```

- Point Pydantic AI, DSPy, or curl examples at `OPENAI_BASE_URL=http://127.0.0.1:8080/v1` , `OPENAI_API_KEY=dummy` , `OPENAI_MODEL=gemma-3-270m-it` .
  - If the download stalls, remove `~/.cache/huggingface/hub/*gemma-3-270m*` and rerun the command (resumes cleanly).
  - For Linux labs without Homebrew, build from source ( `cmake .. && make -j` ) or grab the prebuilt binaries from the llama.cpp releases page.
- 

## "Error: database is locked"

**Symptom:** SQLite file locked during concurrent access

**Cause:** Multiple processes accessing `movies.db` simultaneously

**Fixes:**

```
Option 1: Stop all competing processes
pkill -f uvicorn
pkill -f pytest
sleep 1

Option 2: Use in-memory SQLite for tests (Session 07 conftest.py)
Change DATABASE_URL to sqlite:///memory:

Option 3: Delete and regenerate
rm movies.db
uv run uvicorn app.main:app --reload
```

---

## "Port 8000 already in use"

**Symptoms:**

```
OSError: [Errno 48] Address already in use
```

**Fixes:**

```
Find and kill process using port 8000
lsof -i :8000

Kill the process (replace PID with actual number)
kill -9 <PID>

Or use a different port
uv run uvicorn app.main:app --reload --port 8001
```

---

## "CORS error: Origin not allowed"

**Symptoms:**

- React/Streamlit frontend gets CORS error in browser console

- Error message mentions "Access-Control-Allow-Origin"

#### Fixes:

```
In app/main.py, ensure CORS middleware is added (Session 06)
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
 CORSMiddleware,
 allow_origins=[
 "http://localhost",
 "http://localhost:3000", # React
 "http://localhost:5173", # Vite dev
 "http://localhost:8501", # Streamlit
],
 allow_credentials=True,
 allow_methods=["*"],
 allow_headers=["*"],
)
```

Restart the API after changes:

```
Stop (Ctrl+C) and restart
uv run uvicorn app.main:app --reload
```

---

## Database Issues

### "No such table: movierow"

#### Symptoms:

```
sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) no such table: movierow
```

**Cause:** Database hasn't been initialized (tables not created)

#### Fixes:

```
In app/main.py, ensure startup event runs (Session 05)
from app.db import init_db
from app import repository

@app.on_event("startup")
def startup() -> None:
 init_db(seed_fn=repository.seed_movies)
```

Then restart the app:

```
uv run uvicorn app.main:app --reload
```

## "Foreign key constraint failed"

### Symptom:

```
sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) FOREIGN KEY constraint failed
```

**Cause:** Trying to add a rating for a movie that doesn't exist

### Fixes:

```
Check movies exist
curl http://localhost:8000/movies

Only add ratings for existing movie IDs
curl -X POST http://localhost:8000/ratings \
 -H "Content-Type: application/json" \
 -d '{"movie_id": 1, "user_id": 42, "score": 5}'
```

---

## Testing Issues

### "pytest: command not found"

#### Fixes:

```
Use uv to run pytest
uv run pytest tests/

Or install pytest globally
pip install pytest
```

---

### "Tests pass locally but fail in Docker"

#### Causes:

- Different database state (Docker uses fresh container)
- Missing dependencies in Docker

#### Fixes:

```
Rebuild image with latest deps
docker build --no-cache -t movie-app .

Or inspect test logs
docker run --rm movie-app uv run pytest -v
```

---

### "FixtureNotFound: conftest.py"

#### Symptom:

```
ERROR: Missing fixture 'temporary_db'
```

**Cause:** `conftest.py` not in `tests/` directory or `conftest.py` is empty

**Fixes:**

```
Verify conftest.py exists (Session 07)
ls tests/conftest.py

Copy from Session 07 docs if missing:
See docs/sessions/session-07.md for conftest.py template
```

---

## FastAPI & uvicorn Issues

### "GET /docs returns 404"

**Symptom:** Visiting `http://localhost:8000/docs` returns "Not Found"

**Cause:** FastAPI app not initialized or error during startup

**Fixes:**

```
Check console output for startup errors
uv run uvicorn app.main:app --reload

If error during @app.on_event("startup"), check:
1. Database file path
2. seed_fn exists
3. Models defined before startup runs
```

---

### "422 Unprocessable Entity"

**Symptoms:**

```
{
 "detail": [
 {
 "loc": ["body", "payload"],
 "msg": "field required",
 "type": "value_error.missing"
 }
]
}
```

**Causes:**

- Missing required fields in JSON
- Wrong field type
- Using `embed=True` requires wrapping in a key

## Fixes for Session 12 tool endpoint:

```
WRONG (missing "payload" wrapper)
curl -X POST http://localhost:8000/tool/recommend-movie \
 -H "Content-Type: application/json" \
 -d '{"user_id": 42, "limit": 5}'

CORRECT (with "payload" wrapper, because embed=True)
curl -X POST http://localhost:8000/tool/recommend-movie \
 -H "Content-Type: application/json" \
 -d '{"payload": {"user_id": 42, "limit": 5}}'
```

---

## Authentication & JWT Issues

### "401 Unauthorized"

#### Symptom:

```
{"detail": "Could not validate credentials"}
```

#### Causes:

1. Missing `Authorization` header
2. Invalid token format (not `Bearer <token>`)
3. Token expired

#### Fixes:

```
1. Get fresh token
TOKEN=$(curl -X POST http://localhost:8000/token \
 -H "Content-Type: application/x-www-form-urlencoded" \
 -d "username=teacher&password=classroom" \
 | jq -r '.access_token')

2. Use correct header format
curl -X POST http://localhost:8000/movies \
 -H "Authorization: Bearer $TOKEN" \
 -H "Content-Type: application/json" \
 -d '{"title": "Test", "year": 2024, "genre": "Drama"}'
```

---

### "403 Forbidden"

#### Symptom:

```
{"detail": "Forbidden"}
```

**Cause:** User role too low (need "editor", have "viewer")

#### Fixes:

```
Login with editor role (teacher)
TOKEN=$(curl -X POST http://localhost:8000/token \
 -H "Content-Type: application/x-www-form-urlencoded" \
 -d "username=teacher&password=classroom" \
 | jq -r '.access_token')

Now request should work
curl -X POST http://localhost:8000/movies \
 -H "Authorization: Bearer $TOKEN" \
 -H "Content-Type: application/json" \
 -d '{"title": "Allowed", "year": 2024, "genre": "Drama"}'
```

---

## Frontend Issues (Streamlit & React)

---

### Streamlit "Failed to load movies"

#### Symptoms:

- Streamlit app shows error: "Failed to load movies: HTTPError"
- App cannot reach backend

#### Fixes:

```
1. Verify backend is running on port 8000
curl http://localhost:8000/health

2. Check API_URL in ui/app.py points to correct port
Should be: API_URL = "http://localhost:8000"

3. Ensure CORS middleware is added to FastAPI (Session 06)

4. Restart Streamlit
uv run streamlit run ui/app.py
```

---

### React app stuck loading

#### Symptoms:

- Page shows "Loading movies..." forever
- No error in browser console

#### Fixes:

```
1. Open browser DevTools (F12) → Network tab
Check if GET /movies is pending/failing

2. Verify CORS allows localhost:5173 (Vite dev server)

3. Check Vite is running
npm run dev
```

```
4. Restart Vite dev server
Press 'q' in npm terminal, then npm run dev again
```

---

## Git & Version Control

---

### "fatal: not a git repository"

#### Fixes:

```
git init
git config user.name "Your Name"
git config user.email "your@email.com"
git add .
git commit -m "Initial commit"
```

---

### ".env accidentally committed"

#### Recovery:

```
Remove from Git history (careful!)
git rm --cached .env
git commit --amend --no-edit

Regenerate secrets
openssl rand -hex 32
Update .env with new SECRET_KEY
```

---

## WSL (Windows Subsystem for Linux)

---

### "Cannot find Docker"

#### Symptom:

```
$ docker ps
Command 'docker' not found
```

#### Fix:

- Ensure Docker Desktop is running on Windows host
- In WSL, it will auto-detect

---

### Files slow to access

**Cause:** Storing project on Windows filesystem (C: drive) instead of WSL filesystem

## Fix:

```
Use WSL filesystem for speed
cd /home/your_username/projects
git clone <repo>
cd eass-8-episode
```

---

## When All Else Fails

### 1. Check logs carefully:

```
uv run uvicorn app.main:app --reload 2>&1 | tee app.log
```

### 2. Post in Discord #helpdesk with:

- Error message (full traceback)
- Command you ran
- OS (macOS/Windows/WSL)
- Steps to reproduce

### 3. Schedule office hours if issue persists

---

## Session-Specific Troubleshooting

Detailed troubleshooting for each session is included in the session docs:

- Session 05: Database issues
- Session 06: CORS and frontend connectivity
- Session 07: Test fixtures and pytest
- Session 09: Background task status
- Session 10: Docker Compose multi-service
- Session 11: JWT token expiration
- Session 12: Tool endpoint schema validation